Docket No.: BLD9-2003-0031-US1

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re application of: Hohensee et al.
Application No.: 10/798,045 Group No.: 2625
Filed: March 11, 2004 Examiner: Lawrence E. Wills
For: SYSTEMS AND METHODS FOR IDENTIFYING COMPLEX TEXT IN A PRESENTATION DATA STREAM

Mail Stop Appeal Brief - Patents
Commissioner for Patents
P. O. Box 1450
Alexandria, VA 22313-1450

APPEAL BRIEF

Sir:

Appellants herewith file an updated Brief in support of their Appeal in the above identified matter in response to a notice of non-compliance dated December 4, 2009.

TABLE OF CONTENTS

i. REAL PARTY IN INTEREST

The real party in interest is InfoPrint Solutions Company LLC., the employer of the inventors at the time of the invention and the assignee of the patent rights in the above-identified matter.

ii. RELATED APPEALS AND INTERFERENCES

No other appeals, interferences, or related applications are known to the Appellants, the Appellants' legal representative, or the Assignee, which will directly affect or be directly affected by or have a bearing on the Board's decision in the pending appeal.

iii. STATUS OF CLAIMS

Claims 1, 3, 5-9, 11, 13-14, 16, 18-22, 24, and 26 stand rejected and remain in the application for consideration on appeal. Claims 2, 4, 10, 12, 15, 17, 23, 25, and 27-39 are cancelled and are not under consideration. The 35 U.S.C. § 103(a) rejection of claims 1, 3, 5-9, 11, 13-14, 16, 18-22, 24, and 26 form the basis of this appeal.

iv. STATUS OF AMENDMENTS

No amendments have been filed since the Office Action dated May 28, 2009.

v. SUMMARY OF THE CLAIMED SUBJECT MATTER

Claim 1 recites a method of controlling downstream processing of Unicode Complex text in a print stream. According to the method, a print stream is received including a section of Unicode complex text (Page 3, lines 1-3). The section of Unicode complex text in the print stream is identified (Page 4, lines 1-6; FIG. 2, element 202). A control parameter is inserted in the print stream before the section of Unicode complex text to modify the print stream, where the control parameter comprises a first parameter and a second parameter (Page 4, lines 1-11; FIG. 2, element 206). The first parameter indicates a type of downstream processing for the section of Unicode complex text identified in the print stream (Page 4, lines 8-11). The second parameter is for enabling or disabling the type of downstream processing of the section of Unicode complex text identified in the print stream (Page 11, lines 11-13). Additionally according to the method, the modified print stream is transmitted for downstream processing (Page 13, line 13 to Page 14 line 9; FIG. 3, element 306).

Claim 7 recites a method for processing Unicode complex text in a print stream. According to the method, a control parameter in the print stream for processing a section of Unicode complex text in the print stream is received. The control parameter comprises a first parameter and a second parameter (Page 4, lines 8-11). The first parameter indicates a type of processing for the section of Unicode complex text (Page 4, lines 8-11). The second parameter indicates if the type of processing for the section of Unicode complex text is enabled or disabled (Page 4, lines 9-11). Additionally according to the method, a determination is made if the type of processing is enabled or disabled (FIG. 3, element 314; Page 14, lines 9-14). If the type of processing is enabled, then the section of Unicode complex text is processed responsive to the type of processing indicated by the first parameter (FIG. 3, element 316; Page 9-14). If the type of processing is disabled, then the section of Unicode complex text is not processed responsive to the type of processing indicated by the first parameter (FIG. 3, element 314).

Claim 14 recites a computer readable medium embodying programmed instructions that, when executed by a computer, perform a method for controlling downstream processing of Unicode complex text in a print stream (FIG. 7; Page 21, line 26 to Page 22 line 11). According to the method, a print stream is received including a

section of Unicode complex text (Page 3, lines 1-3). The section of Unicode complex text in the print stream is identified (Page 4, lines 1-6; FIG. 2, element 202). A control parameter is inserted in the print stream before the section of Unicode complex text to modify the print stream, where the control parameter comprises a first parameter and a second parameter (Page 4, lines 1-11; FIG. 2, element 206). The first parameter indicates a type of downstream processing for the section of Unicode complex text identified in the print stream (Page 4, lines 8-11). The second parameter is for enabling or disabling the type of downstream processing of the section of Unicode complex text identified in the print stream (Page 11, lines 11-13). Additionally according to the method, the modified print stream is transmitted for downstream processing (Page 13, line 13 to Page 14 line 9; FIG. 3, element 306).

Claim 20 recites a computer readable medium embodying programmed instructions that, when executed by a computer, perform a method processing Unicode complex text in a print stream (FIG. 7; Page 21, line 26 to Page 22 line 11). According to the method, a control parameter in the print stream for processing a section of Unicode complex text in the print stream is received. The control parameter comprises a first parameter and a second parameter (Page 4, lines 8-11). The first parameter indicates a type of processing for the section of Unicode complex text (Page 4, lines 8-11). The second parameter indicates if the type of processing for the section of Unicode complex text is enabled or disabled (Page 4, lines 9-11). Additionally according to the method, a determination is made if the type of processing is enabled or disabled (FIG. 3, element 314; Page 14, lines 9-14). If the type of processing is enabled, then the section of Unicode complex text is processed responsive to the type of processing indicated by the first parameter (FIG. 3, element 316; Page 9-14). If the type of processing is disabled, then the section of Unicode complex text is not processed responsive to the type of processing indicated by the first parameter (FIG. 3, element 314).

vi. GROUNDS OF REJECTION TO BE REVIEWED ON APPEAL

1. Whether claims 1, 3, 5-9, 11, 13-14, 16, 18-22, 24, and 26 are unpatentable over U.S. Patent No.: RE37,258 (Patel) in view of U.S. PG-PUB 2003/0023590 (Atkin) under 35 U.S.C. § 103(a).

vii. ARGUMENT

1. Rejection of claims 1, 3, 5-9, 11, 13-14, 16, 18-22, 24, and 26 under 35 U.S.C. § 103(a).

The Examiner has rejected claims 1, 3, 5-9, 11, 13, 14, 16, 18-22, 24, and 26 under 35 U.S.C. § 103(a) as being unpatentable over U.S. Patent Number RE37,258 (Patel) in view of U.S. PG-PUB 2003/0023590 (Atkin). The Appellants submit that the claims are non-obvious in view of the combination of Patel and Atkin.

A print stream contains various representations of data to be printed. For example, the print stream may contain pictures, simple text, or in some cases, complex text such as Unicode complex text. Unicode complex text is an encoding standard used to represent many different character sets and glyphs (pictorial represented languages, such as Chinese or Hebrew). Some languages are printed and read from left to right, and others are printed and read from right to left. In some cases, portions of languages require that the left to right or right to left orientations change depending on what characters are being printed. In the case of Unicode complex text, this is accomplished by processing the Unicode complex text print stream to modify the character placements on a page layout. Because processing the Unicode complex print stream requires computational effort, it may be desirable to disable this processing for print draft or print proofing operations to increase the print speed.

The Appellants submit that Atkin does not teach or reasonably suggest the limitation of "a second parameter for disabling the type of downstream processing of the section of Unicode complex text identified in the print stream" as recited in claim 1. The Examiner suggests that Atkin teaches that 'tag' values included in a Unicode datastream may be used to disable the downstream processing of a section of Unicode text. Specifically, the Examiner suggests that a 'cancel' tag can be used to disable the downstream processing. The Appellants respectfully disagree.

Atkin discloses an approach for including metadata within a Unicode datastream by introducing 97 new characters to the Unicode specification, called 'tags'. The 97 new characters would only be used to indicate embedded metadata within the datastream, thus removing any 'overloading' of the original Unicode characters. When characters are

'overloaded', determining whether a character is data or metadata depends on the context in which the character is found. The inclusion of the 97 new characters in the Unicode specification allows for the construction of simple parsers for separating the metadata from the data since there is no 'overloading' of characters (Paragraph 40).

One example disclosed in Atkin for including metadata in a serial Unicode datastream is shown in table 4. In table 4, a language tag marks the beginning and the end of a section of Unicode data in the datastream, which indicates that the section of text is in the French Language. Atkin further discloses that the language tag remains in scope (i.e., applies to subsequently received characters in the datastream) until a cancel tag is received. In other words, when a parser receives the French language tag in the datastream, subsequently received characters in the datastream are also included in the French language metadata assignment until the cancel tag is received.

Although Atkin suggests that the cancel tag can be used mark the end of a section of characters in the datastream assigned language metadata, there is no teaching or suggestion in Atkin that the cancel tag is operable to disable language metadata assigned to the section of characters. To relate this to the metadata language example above, a section of the datastream is identified as residing between the language tag and the cancel tag. Characters in the identified section of the datastream are assigned the French language metadata irrespective of the inclusion of the cancel marking the end of the identified section.

The Appellants further submit that the method of claim 1 recites advantages over Atkin. In claim 1, control parameters can be inserted in a print stream to enable and disable downstream processing of specific sections of complex text within the datastream. For example, the control parameter may indicate to a printer to skip processing of the specific sections of the complex text to increase the speed of rendering the print stream to paper, such as when generating a draft. Subsequently, the print stream may be regenerated such that the control parameter indicates to the printer to process the specific sections of the complex text for generating a final output. The Appellants maintain that the cancel tag in Atkin does not teach or suggest this functionality.

The Examiner has further indicated that Patel fails to teach a parameter indicating a type of downstream processing and a parameter enabling or disabling processing, of

which the Appellants agree. The Appellants therefore submit that claim 1 is non-obvious in view of the cited art for at least the reasons provided above. Independent claims 7, 14, and 20 are non-obvious for at least the same reasons. Dependent claims 3, 5-6, 8-9, 11, 13, 16, 18-19, 21-22, 24, and 26 are non-obvious for at least depending on base claims 1, 7, 14, and 20.

viii. CLAIMS APPENDIX

1. A method of controlling downstream processing of Unicode complex text in a print stream, the method comprising:

receiving the print stream including a section of Unicode complex text;

identifying the section of the Unicode complex text in the print stream;

inserting a control parameter in the print stream before the section of the Unicode complex text to modify the print stream, wherein the control parameter comprises:

a first parameter indicating a type of downstream processing for the section of the Unicode complex text identified in the print stream; and

a second parameter for enabling and disabling the type of downstream processing of the section of the Unicode complex text identified in the print stream; and

transmitting the modified print stream for downstream processing.

3. The method of claim 1 wherein the first parameter indicates bidirectional (bidi) layout processing of the Unicode complex text.

5. The method of claim 1 wherein the first parameter indicates layout processing of glyphs within the Unicode complex text.

6. The method of claim 1 wherein the control parameter further includes a third parameter indicating text positioning at the completion of the processing of the Unicode complex text.

7. A method for processing Unicode complex text in a print stream, the method comprising:

receiving a control parameter in the print stream for processing a section of Unicode complex text in the print stream, wherein the control parameter comprises:

a first parameter indicating a type of processing for the section of the Unicode complex text; and

a second parameter indicating if the type of processing for the section of the Unicode complex text is enabled or disabled;

determining if the type of processing is enabled or disabled;

processing the section of the Unicode complex text responsive to the type of processing indicated by the first parameter and the determination that the type of processing is enabled; and

not processing the section of the Unicode complex text responsive to the type of processing indicated by the first parameter and the determination that the type of processing is disabled.

8. The method of claim 7 wherein the first parameter indicates bidirectional (bidi) layout processing of the Unicode complex text.

9. The method of claim 8 wherein the first parameter indicates a paragraph direction for the bidirectional layout processing of the Unicode complex text.

11. The method of claim 7 wherein the first parameter indicates layout processing of glyphs within the Unicode complex text.

13. The method of claim 7 wherein the control parameter further includes a third parameter indicating text positioning at the completion of the downstream processing of the Unicode complex text.

14. A computer readable medium embodying programmed instructions that, when executed by a computer, performs a method for controlling downstream processing of Unicode complex text in a print stream, the method comprising:

receiving the print stream including a section of Unicode complex text;

identifying the section of the Unicode complex text in the print stream;

inserting a control parameter in the print stream before the section of the Unicode complex text to modify the print stream, wherein the control parameter comprises:

a first parameter indicating a type of downstream processing for the section of the Unicode complex text identified in the print stream; and

a second parameter for enabling and disabling the type of downstream processing of the section of the Unicode complex text identified in the print stream; and

transmitting the modified print stream for downstream processing.

16. The computer readable medium of claim 14 wherein the first parameter indicates bidirectional (bidi) layout processing of the Unicode complex text.

18. The computer readable medium of claim 14 wherein the first parameter indicates layout processing of glyphs within the Unicode complex text.

19. The computer readable medium of claim 14 wherein the control parameter further includes a third parameter indicating text positioning at the completion of the downstream processing of the Unicode complex text.

20 A computer readable medium embodying programmed instructions that, when executed by a computer, performs a method for processing Unicode complex text in a print stream, the method comprising:

receiving a control parameter in the print stream for processing a section of Unicode complex text in the print stream, wherein the control parameter comprises:

a first parameter indicating a type of processing for the section of the Unicode complex text; and

a second parameter indicating if the type of processing for the section of the Unicode complex text is enabled or disabled;

determining if the type of processing is enabled or disabled;

processing the section of the Unicode complex text responsive to the type of processing indicated by the first parameter and the determination that the type of processing is enabled; and

not processing the section of the Unicode complex text responsive to the type of processing indicated by the first parameter and the determination that the type of processing is disabled.

21. The computer readable medium of claim 20 wherein the first parameter indicates bidirectional (bidi) layout processing of the Unicode complex text.

22. The computer readable medium of claim 21 wherein the first parameter indicates a paragraph direction for the bidirectional layout processing of the Unicode complex text.

24. The computer readable medium of claim 20 wherein the first parameter indicates layout processing of glyphs within the Unicode complex text.

26. The computer readable medium of claim 20 wherein the control parameter further includes a third parameter indicating text positioning at the completion of processing of the Unicode complex text.

xi. EVIDENCE APPENDIX

Included are copies of the evidence relied upon by the Examiner as to the grounds of the rejections under 35 U.S.C. § 103(a) to be reviewed on appeal.

1. U.S. Patent No.: RE37,258 (Patel).

2. U.S. PG-PUB 2003/0023590 (Atkin).

x. RELATED PROCEEDINGS APPENDIX

None.

SUMMARY

Appellants argue that the Examiner's rejections of claims 1, 3, 5-9, 11, 13-14, 16, 18-22, 24, and 26 under 35 U.S.C. § 103(a) are inadequate as a matter of law and should be reversed.

Date: December 7, 2009

/Sean J. Varley/
SIGNATURE OF PRACTITIONER

Sean J. Varley, Reg. No. 62,397
Duft Bornsen & Fishman, LLP
Telephone: (303) 786-7687
Facsimile: (303) 786-7691
Customer No.: 50441

US00RE37258E

(19) **United States**

(12) **Reissued Patent**

Patel et al.

(10) **Patent Number: US RE37,258 E**

(45) **Date of Reissued Patent: Jul. 3, 2001**

(54) **OBJECT ORIENTED PRINTING SYSTEM**

(75) Inventors: **Jayendra Natubhai Patel**, Sunnyvale; **Ryoji Watanabe**, Cupertino; **Mark Peek**, Lomond; **L. Bayles Holt**, San Jose; **Mahinda K. de Silva**, Mountain View, all of CA (US)

(73) Assignee: **Object Technology Licensing Corp.**, Cupertino, CA (US)

(21) Appl. No.: **09/173,465**

(22) Filed: **Oct. 14, 1998**

Related U.S. Patent Documents

Reissue of:

(64) Patent No.: **5,566,278**
Issued: **Oct. 15, 1996**
Appl. No.: **08/111,238**
Filed: **Aug. 24, 1993**

(51) **Int. Cl.**[7] **G06F 15/00**

(52) **U.S. Cl.** **358/1.15**; 358/1.13

(58) **Field of Search** 358/1.13, 1.15, 358/1.18, 1.1, 1.16, 1.17; 345/433, 438; 710/1, 11, 14, 19, 24, 33, 34, 39, 50, 54, 72, 100, 104, 105, 106, 112, 129; 395/500.44

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,821,220	*	4/1989	Duisberg	364/578
4,885,717	*	12/1989	Beck et al.	364/900
4,891,630	*	1/1990	Friedman et al.	340/706
4,953,080	*	8/1990	Dysart et al.	364/200
5,025,398	*	6/1991	Nelson	395/112
5,025,399	*	6/1991	Wendt et al.	395/117
5,041,992	*	8/1991	Cunningham et al.	364/518
5,050,090	*	9/1991	Golub et al.	364/478
5,052,834	*	10/1991	Feistel et al.	400/121
5,060,276	*	10/1991	Morris et al.	382/8
5,075,848	*	12/1991	Lai et al.	395/425
5,093,914	*	3/1992	Coplien et al.	395/700
5,119,475	*	6/1992	Smith et al.	395/156
5,123,757	*	6/1992	Nagaoka et al.	395/114
5,125,091	*	6/1992	Staas, Jr. et al.	395/650
5,133,075	*	7/1992	Risch	395/800
5,136,705	*	8/1992	Stubbs et al.	395/575
5,151,987	*	9/1992	Abraham et al.	395/575
5,181,162	*	1/1993	Smith et al.	364/419
5,226,112	*	7/1993	Mensing et al.	395/114
5,287,194	*	2/1994	Lobiondo	395/114
5,303,336	*	4/1994	Kageyama et al.	395/114
5,323,393	*	6/1994	Barrett et al.	370/85.8
5,337,258	*	8/1994	Dennis	395/575
5,353,388	*	10/1994	Motoyama	395/117
5,495,561	*	2/1996	Holt	395/112

OTHER PUBLICATIONS

IBM Technical Disclosure Bulletin, V. 34(10A), New York, US, pp. 912–193 "Use of Agraphical User Interface for Printers" Mar. 1992 .*

* cited by examiner

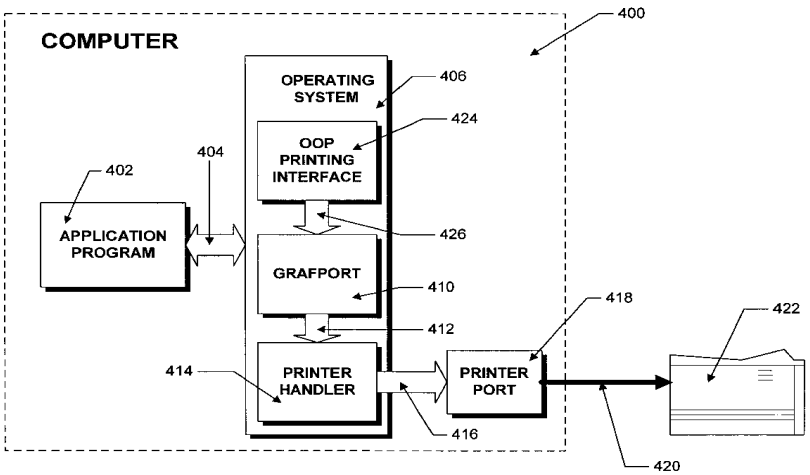
Primary Examiner—Dov Popovici

(74) *Attorney, Agent, or Firm*—Morgan & Finnegan, LLP

(57) **ABSTRACT**

An object-oriented printing system includes objects that provide query, data transfer, and control methods. The inventive object-oriented printing system communicates with the remainder of the operating system by means of a standard interface such as a grafport and printer drivers are provided for each printer type within the operating system. Thus, an application not only need not worry about the particular printer/computer combination with which it is to operate, but also need not have a built in document formatting capability. The printing system includes objects that provide queries for device identification, optimized imaging, and printer status. Other objects are also provided for data transfer to bracket connections prior to sending and receiving information. Still other objects are provided for cancelling a print job, pausing a job, and clearing out a job. Finally, an object is also provided for supporting multiple streams of communication to an imaging task.

33 Claims, 12 Drawing Sheets



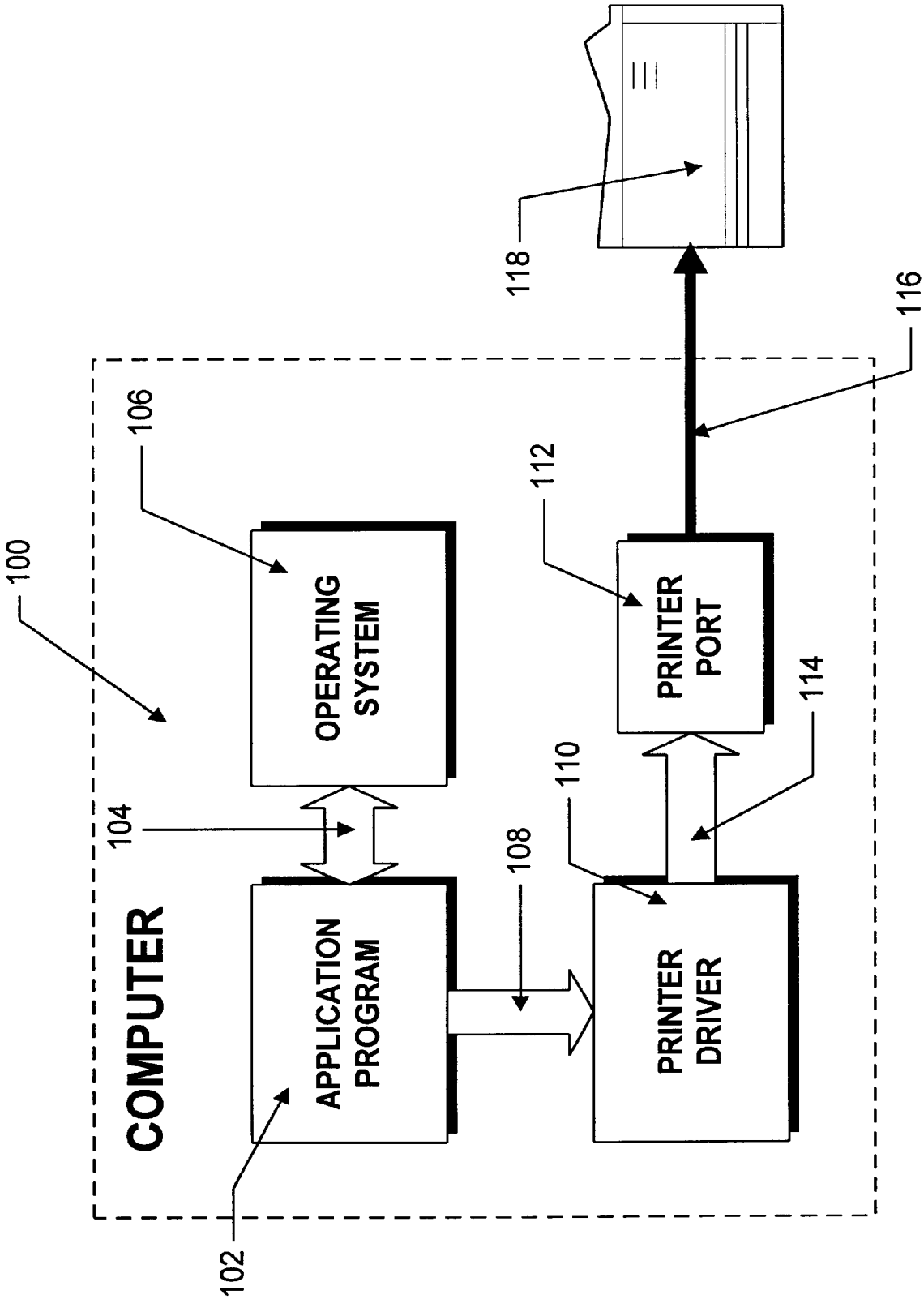


FIG. 1 (PRIOR ART)

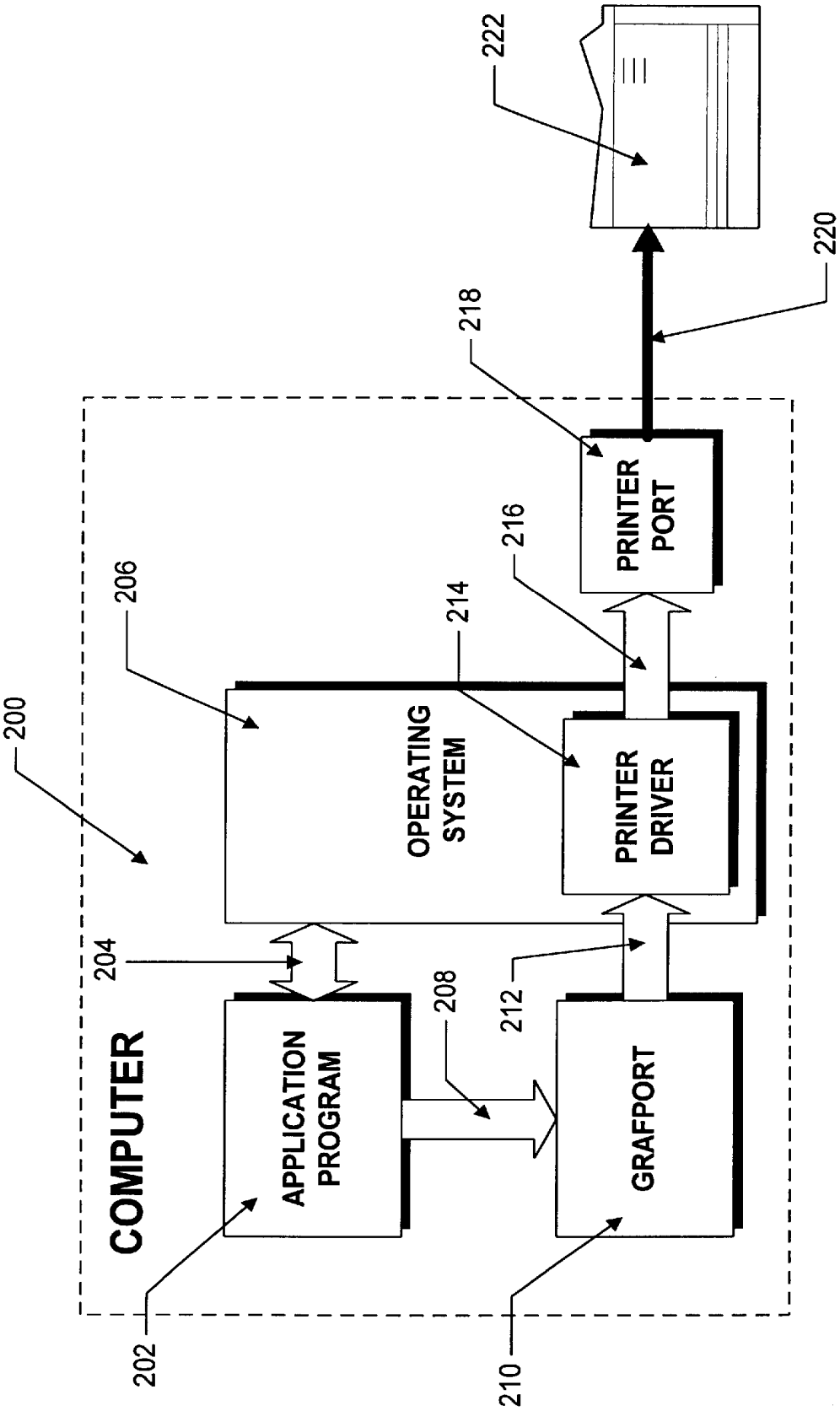


FIG. 2 (PRIOR ART)

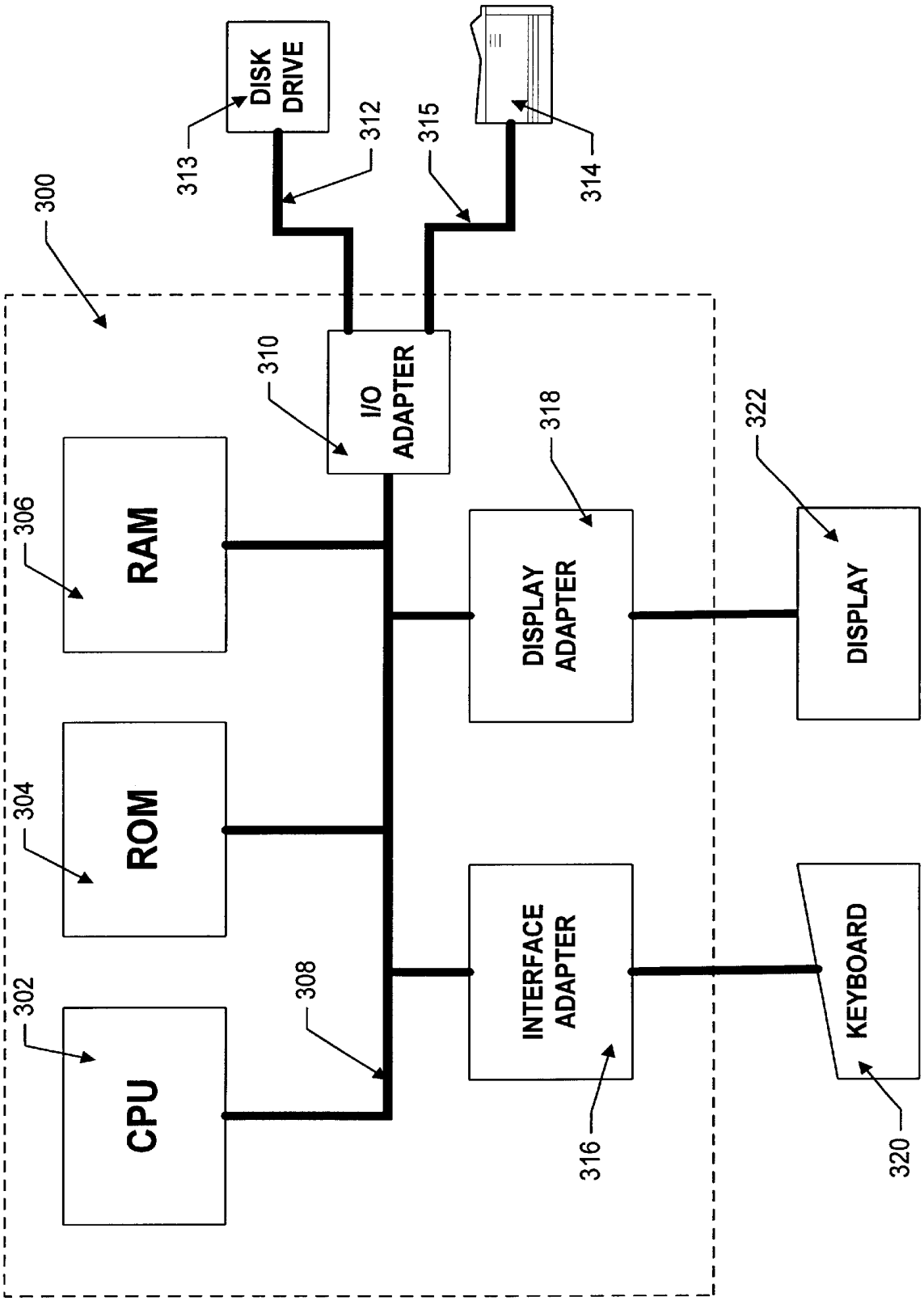

DISK DRIVE
313
312
315
314
300
310
I/O ADAPTER
318
322
306
RAM
DISPLAY ADAPTER
DISPLAY
304
ROM
302
308
INTERFACE ADAPTER
KEYBOARD
CPU
316
320

FIG. 3

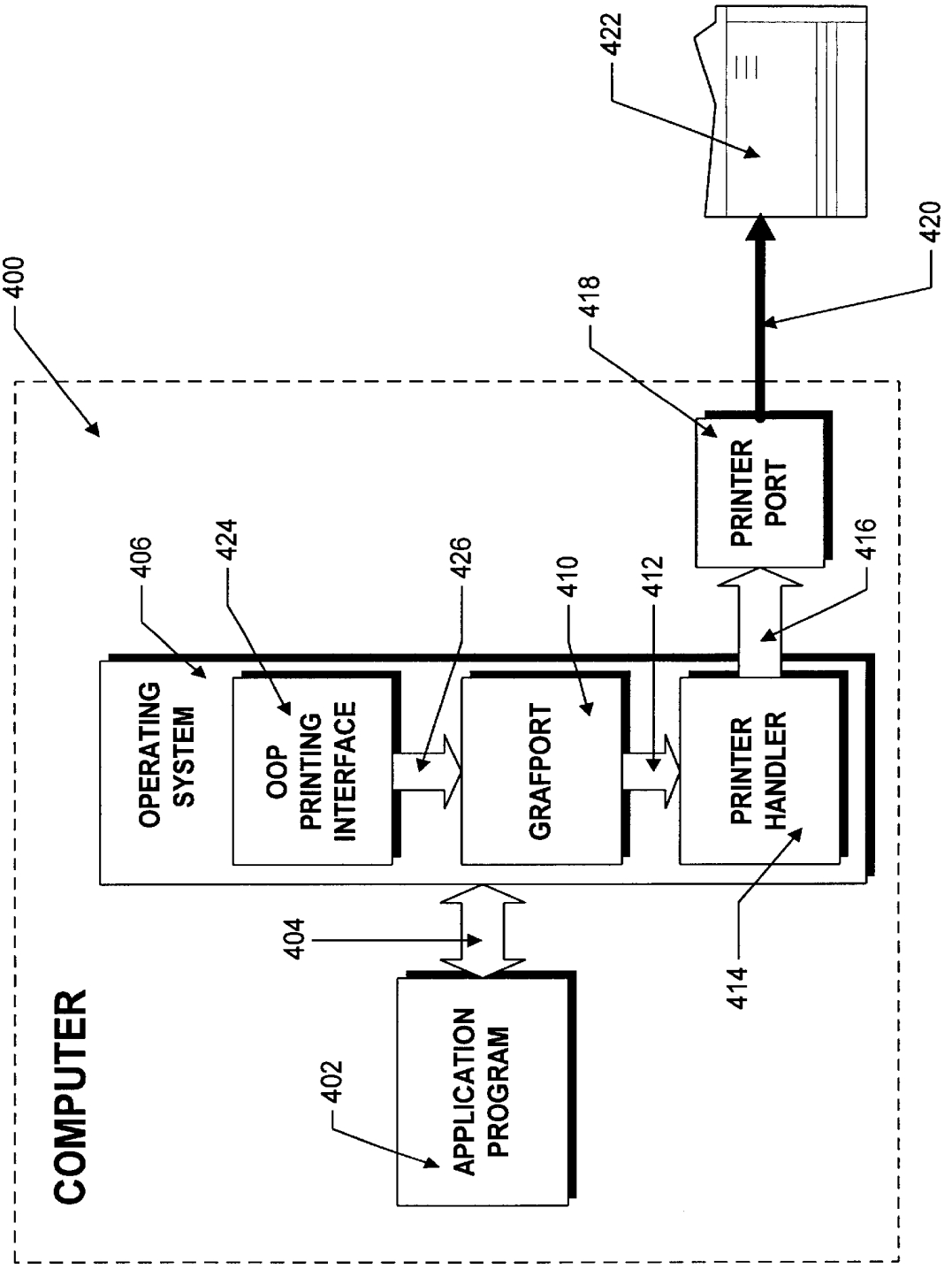

400
COMPUTER
APPLICATION PROGRAM
402
404
OPERATING SYSTEM
406
OOP PRINTING INTERFACE
424
GRAFPORT
410
PRINTER HANDLER
412
414
PRINTER PORT
416
418
420
422
426

FIG. 4

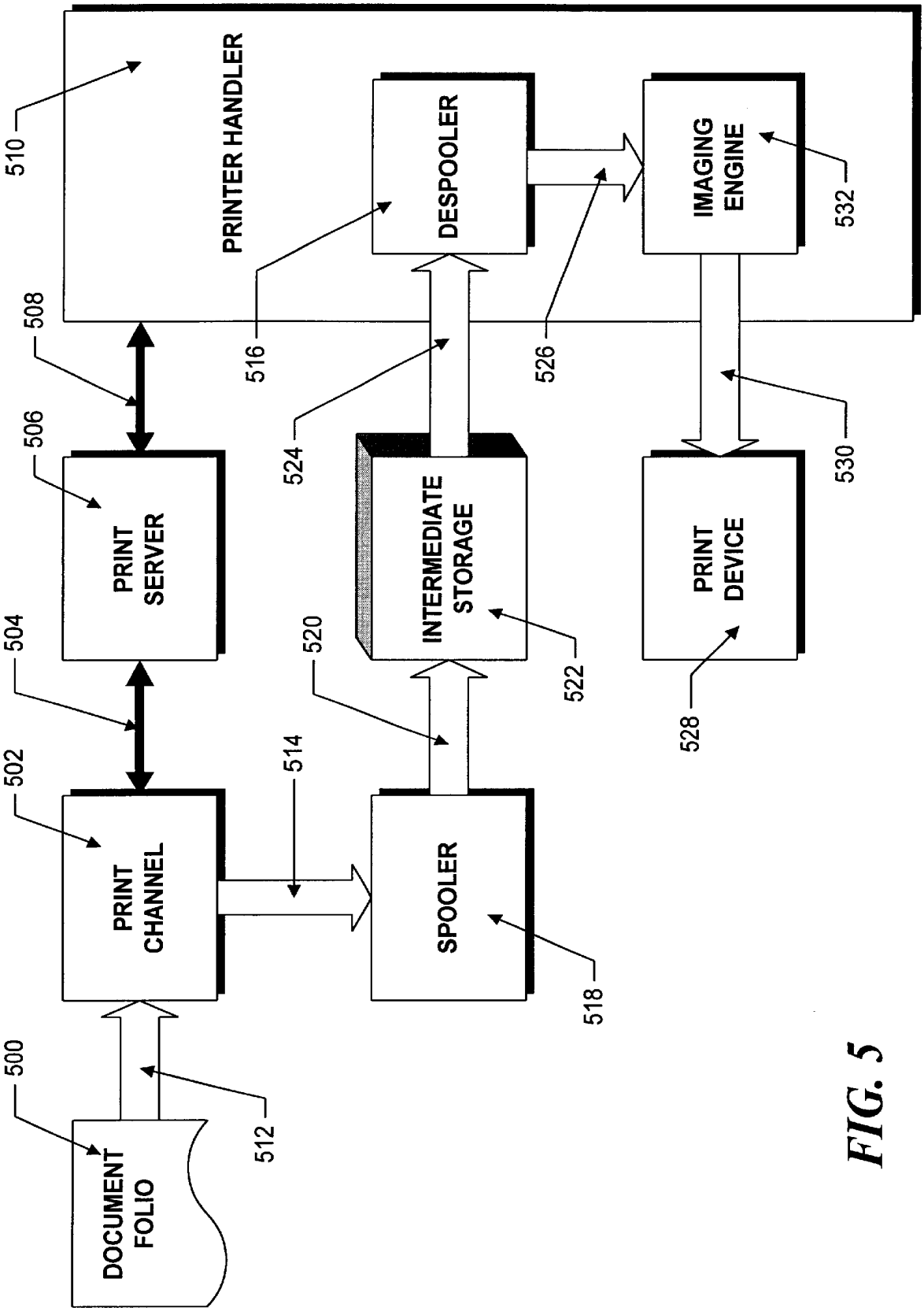

PRINTER HANDLER
510
DESPOOLER
IMAGING ENGINE
532
508
516
526
506
PRINT SERVER
524
INTERMEDIATE STORAGE
PRINT DEVICE
530
504
520
522
528
502
PRINT CHANNEL
514
SPOOLER
518
500
DOCUMENT FOLIO
512

FIG. 5

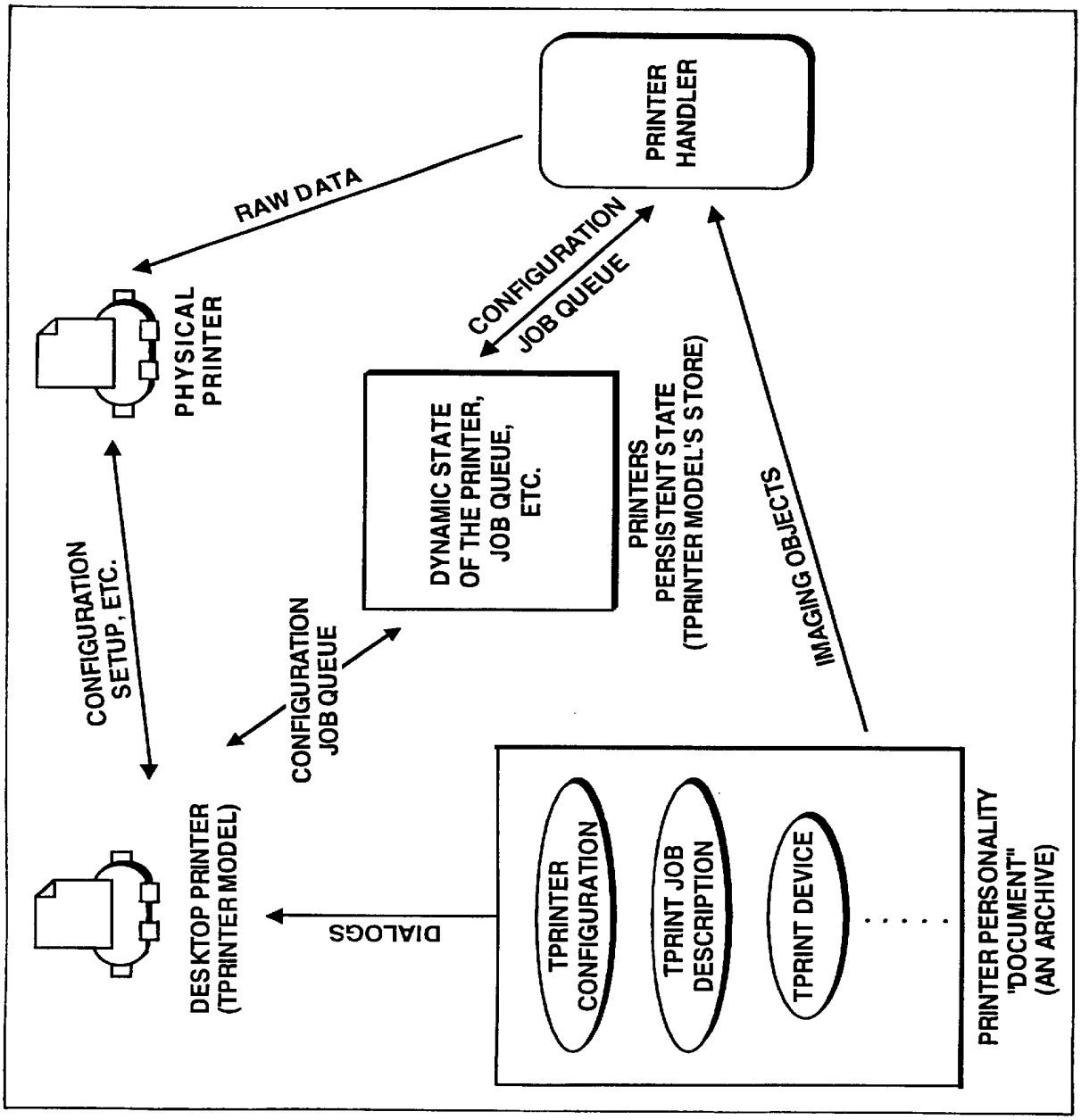

PRINTER HANDLER
RAW DATA
PHYSICAL PRINTER
CONFIGURATION
JOB QUEUE
DYNAMIC STATE OF THE PRINTER, JOB QUEUE, ETC.
PRINTERS PERSISTENT STATE (TPRINTER MODEL'S STORE)
CONFIGURATION SETUP, ETC.
CONFIGURATION JOB QUEUE
IMAGING OBJECTS
DESKTOP PRINTER (TPRINTER MODEL)
DIALOGS
TPRINTER CONFIGURATION
TPRINT JOB DESCRIPTION
TPRINT DEVICE
. . . . .
PRINTER PERSONALITY "DOCUMENT" (AN ARCHIVE)

FIGURE 6

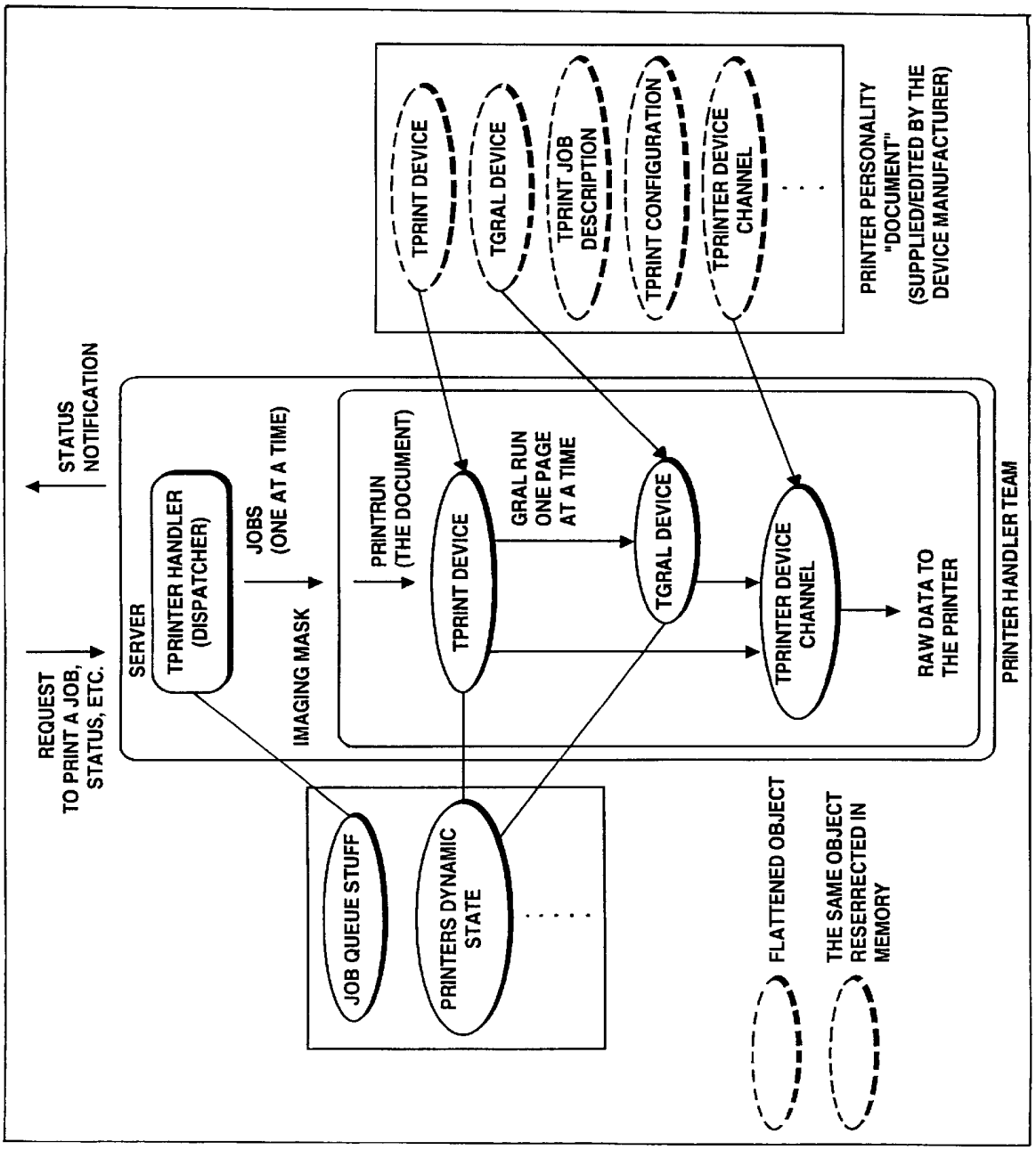

REQUEST
TO PRINT A JOB,
STATUS, ETC.
STATUS
NOTIFICATION
SERVER
TPRINTER HANDLER
(DISPATCHER)
JOBS
(ONE AT A TIME)
IMAGING MASK
PRINTRUN
(THE DOCUMENT)
TPRINT DEVICE
GRAL RUN
ONE PAGE
AT A TIME
TGRAL DEVICE
TPRINTER DEVICE
CHANNEL
RAW DATA TO
THE PRINTER
PRINTER HANDLER TEAM
TPRINT DEVICE
TGRAL DEVICE
TPRINT JOB
DESCRIPTION
TPRINT CONFIGURATION
TPRINTER DEVICE
CHANNEL
PRINTER PERSONALITY
"DOCUMENT"
(SUPPLIED/EDITED BY THE
DEVICE MANUFACTURER)
JOB QUEUE STUFF
PRINTERS DYNAMIC
STATE
FLATTENED OBJECT
THE SAME OBJECT
RESERRECTED IN
MEMORY

FIGURE 7

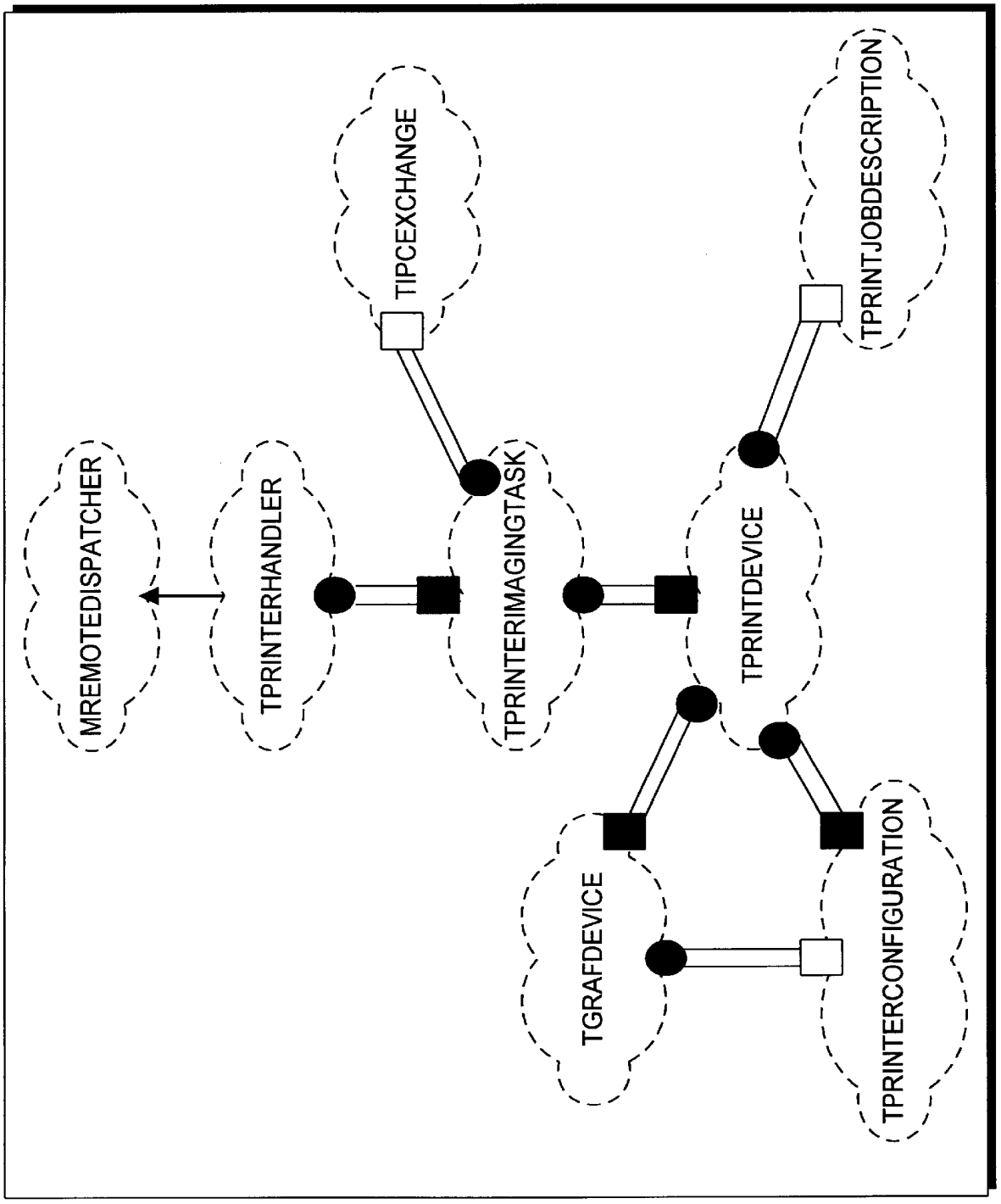

MREMOTEDISPATCHER
TPRINTERHANDLER
TIPCEXCHANGE
TPRINTERIMAGINGTASK
TPRINTJOBDESCRIPTION
TPRINTDEVICE
TGRAFDEVICE
TPRINTERCONFIGURATION

FIG. 8

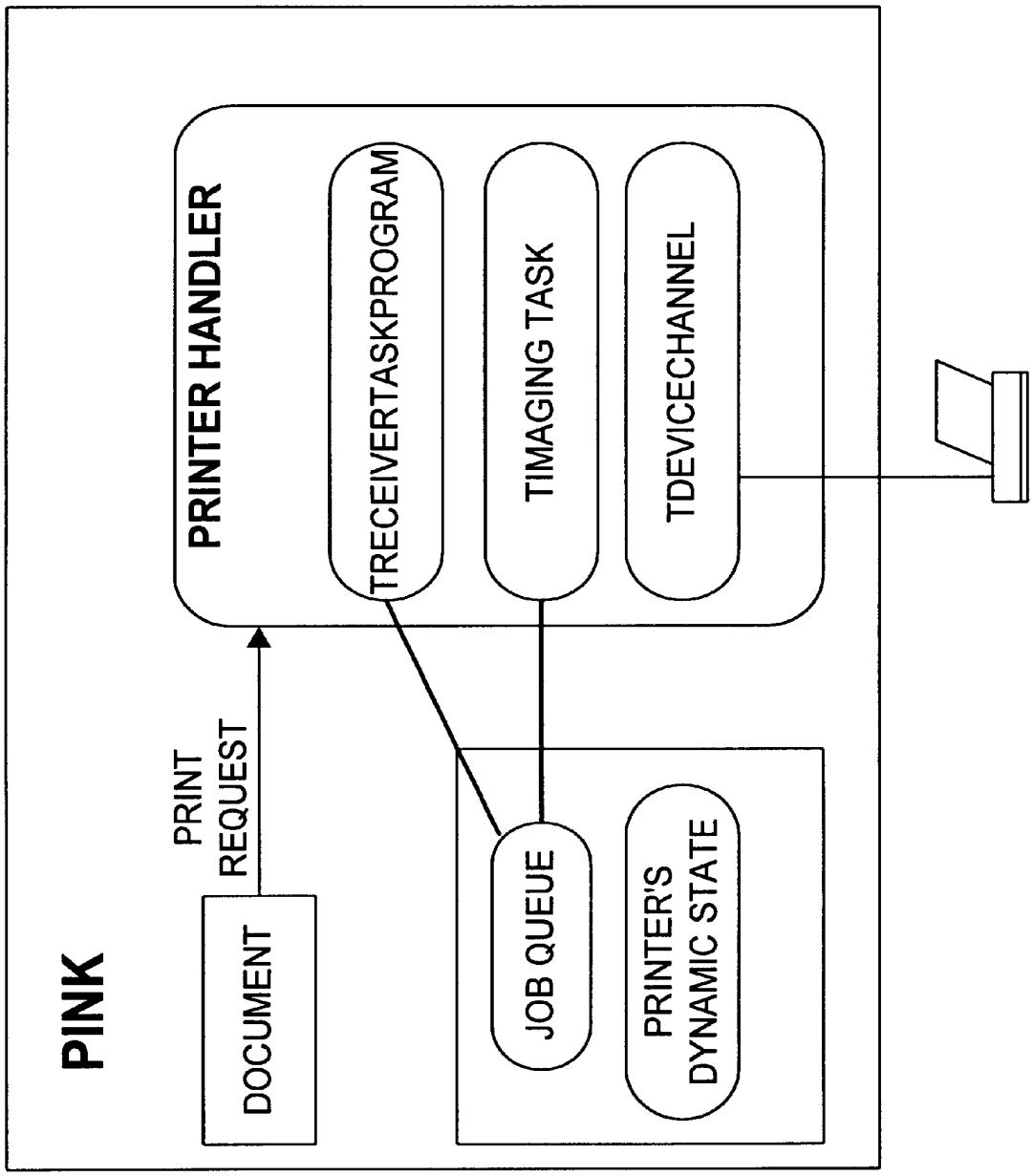

PINK
DOCUMENT
PRINT REQUEST
PRINTER HANDLER
TRECEIVERTASKPROGRAM
TIMAGING TASK
TDEVICECHANNEL
JOB QUEUE
PRINTER'S DYNAMIC STATE

FIG. 9

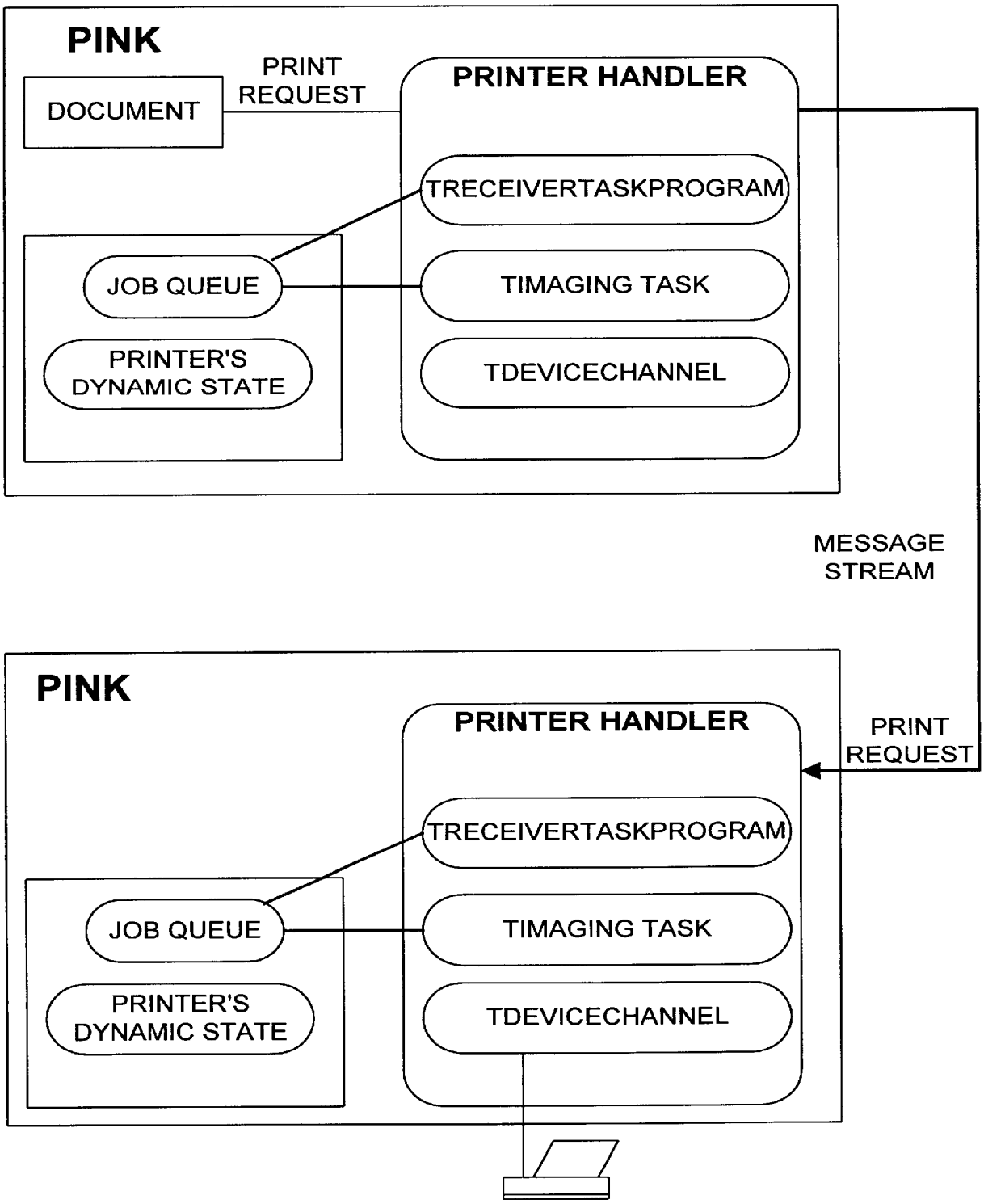

PINK
DOCUMENT
PRINT REQUEST
PRINTER HANDLER
TRECEIVERTASKPROGRAM
JOB QUEUE
TIMAGING TASK
PRINTER'S DYNAMIC STATE
TDEVICECHANNEL
MESSAGE STREAM
PINK
PRINTER HANDLER
PRINT REQUEST
TRECEIVERTASKPROGRAM
JOB QUEUE
TIMAGING TASK
PRINTER'S DYNAMIC STATE
TDEVICECHANNEL

FIG. 10

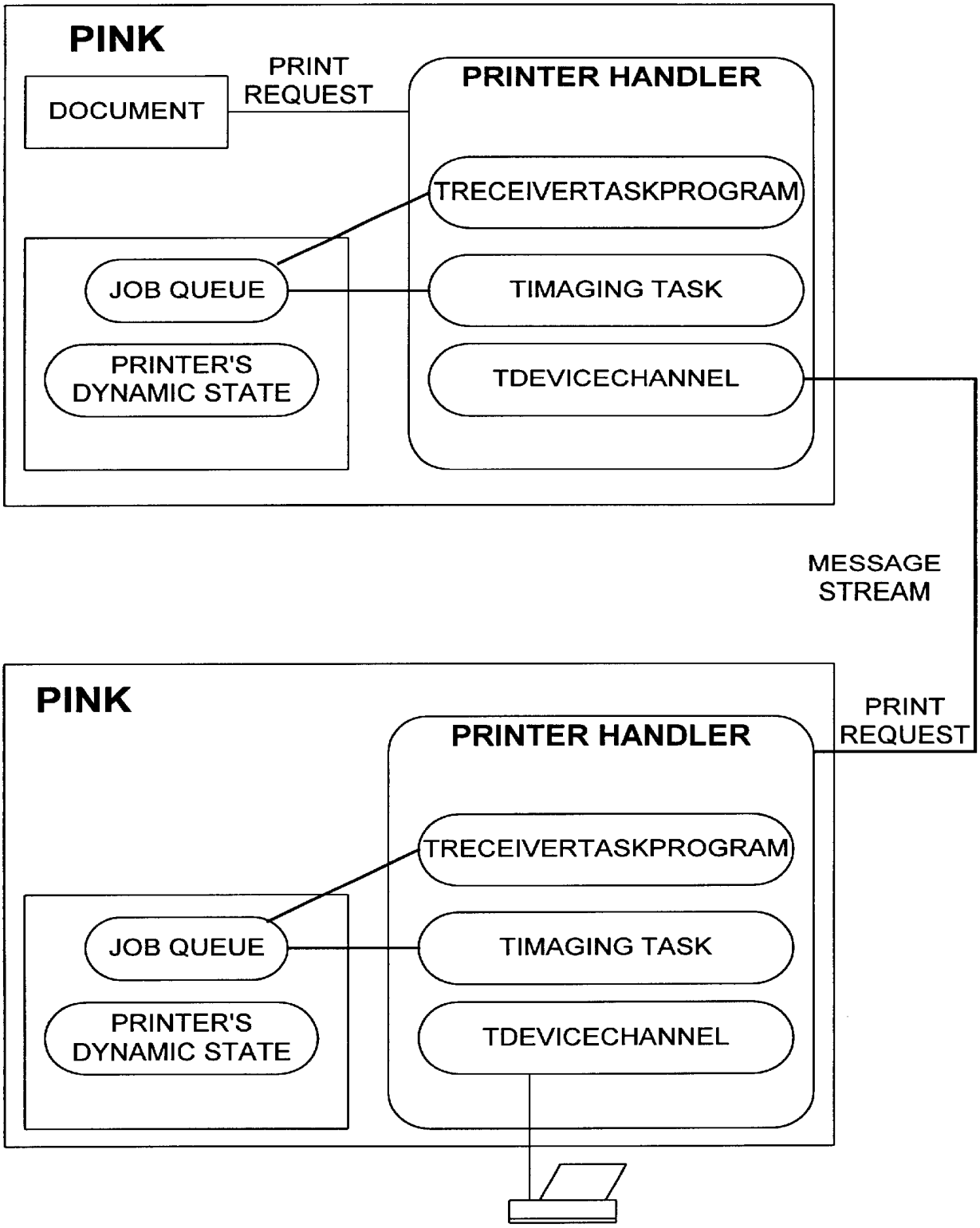

PINK
DOCUMENT
PRINT REQUEST
PRINTER HANDLER
TRECEIVERTASKPROGRAM
JOB QUEUE
TIMAGING TASK
PRINTER'S DYNAMIC STATE
TDEVICECHANNEL
MESSAGE STREAM
PINK
PRINT REQUEST
PRINTER HANDLER
TRECEIVERTASKPROGRAM
JOB QUEUE
TIMAGING TASK
PRINTER'S DYNAMIC STATE
TDEVICECHANNEL

FIG. 11

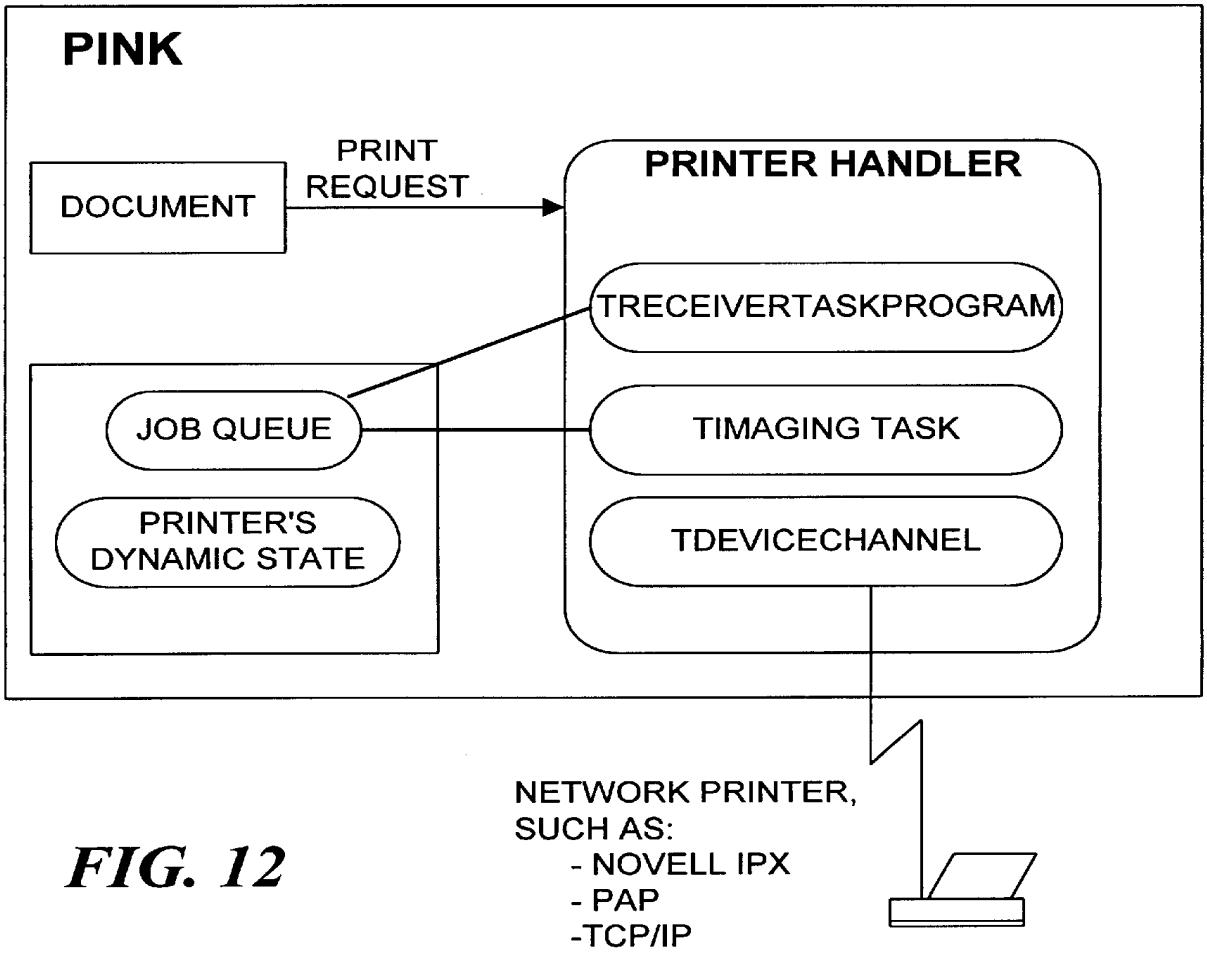

PINK
DOCUMENT
PRINT
REQUEST
PRINTER HANDLER
TRECEIVERTASKPROGRAM
JOB QUEUE
TIMAGING TASK
PRINTER'S
DYNAMIC STATE
TDEVICECHANNEL
NETWORK PRINTER,
SUCH AS:
- NOVELL IPX
- PAP
-TCP/IP

FIG. 12

OBJECT ORIENTED PRINTING SYSTEM

Matter enclosed in heavy brackets [ ] appears in the original patent but forms no part of this reissue specification; matter printed in italics indicates the additions made by reissue.

COPYRIGHT NOTIFICATION

FIELD OF THE INVENTION

The invention generally relates to improvements in computer systems and, more particularly, to operating system software for printing documents.

BACKGROUND OF THE INVENTION

One of the most important aspects of a modern computer system is the ability to generate a "hard" copy of textual information or graphics which can be manipulated by the computer, visually displayed and stored. In order to accomplish this task, a computer system generally includes a printing device which is electrically connected to the computer system and controlled by it in order to generate a permanent image on a selected medium. Examples of printing devices in common use are dot matrix, ink jet and laser printers which fix permanent images on paper under control of the computer. Although paper is the most common medium, other media are often used, such as photographic film.

In order to print a document which is displayed on the monitor or stored, within the memory, several actions must take place. First, since the print medium generally has a fixed size, the printable information must be divided into pieces which are small enough to fit on the selected medium, a process which is called pagination. In addition, the information may need to be reformatted from the format in which it is either displayed or stored into a format which is suitable for controlling the printing device to actually perform the printing on the medium. The reformatting in this latter step may include the insertion of control commands into the printable information in order to control the printing device. These added commands may, for example, include such commands as carriage returns, line feeds, form feeds, spaces and font information, all of which format the printable information. The reformatting may also include a conversion step in which a graphical display is converted into the form used by the printing device.

The pagination and reformatting necessary to convert the printable information into a form which can be printed on a given printing device is generally performed by software programs running within the computer system. Software programs operating on a computing system generally can be categorized into two broad classes: operating systems which are usually specific to a type of computer system and consist of a collection of utility programs that enable the computer system to perform basic operations, such as storing and retrieving information on a peripheral disk memory, displaying information on an associated video display, performing rudimentary file operations including the creation, naming and renaming of files and, in some cases, performing diagnostic operations in order to discover or recover from malfunctions.

By itself, the operating system generally provides only very basic functions and must be accompanied by an "application" program. The application program interacts with the operating system to provide much higher level functionality and a direct interface with the user. During the interaction between the application program and the operating system, the application program typically makes use of operating system functions by sending a series of task commands to the operating system which then performs the requested tasks. For example, the application program may request that the operating system store particular information on the computer disk memory or display information on the video display.

FIG. **1** is a schematic illustration of a typical computer system utilizing both an application program and an operating system. The computer system is schematically represented by dotted box **100**, the application program is represented by box **102** and the operating system by box **106**. The interaction between the application program **102** and the operating system **106** is illustrated schematically by arrow **104**. This dual program system is used on many types of computers systems ranging from mainframes to personal computers.

The method for handling printing, however, varies from computer to computer, and, in this regard, FIG. **1** represents a prior art personal computer system. In order to provide printing functions, the application program **102** interacts (as shown schematically by arrow **108**) with printer driver software **110**. Printer driver software **110** is generally associated with an application program and reformats and converts the printable information as necessary. Since each printer has its own particular format and control command set, which must be embedded in the text properly to control the printer, the printer driver software **110** must be specifically designed to operate with one printer or one type of printer.

The printer driver software **110** produces a reformatted information stream containing the embedded commands as shown schematically as arrow **114**. The converted information stream is, in turn, applied to a printer port **112** which contains circuitry that converts the incoming information stream into electrical signals. The signals are, in turn, sent over a cable **116** to the printer **118**. Printer **118** usually contains an "imaging engine" which is a hardware device or a ROM-programmed computer which takes the incoming information stream and converts it into the electrical signals necessary to drive the actual printing elements. The result is a "hard copy" output on the selected medium.

While the configuration shown in FIG. **1** generally works in a satisfactory manner, it has several drawbacks. Since the printer driver software **110** is specific to each type of printer, a separate driver had to be provided for each printer type with which the application program is to operate. In the personal computer market, there are a large number of different printer types that are compatible with each type of computer and, therefore, as the number of printer types proliferated, so did the number of printer drivers which were required for each application program so that the program was compatible with most available printers. Therefore, application program developers had to provide larger and larger numbers of printer drivers with each application program, resulting in wasted time and effort and wasted disk space to hold the drivers, only one or two of which were of interest to any particular user. Unfortunately, if a user purchased an application program and it did not include a printer driver which could control the printer which the user owned, unpredictable operation occurred, resulting in program returns and user dissatisfaction.

In addition, it was also necessary for each application program to provide high level printing functions such as pagination and page composition (including addition of margins, footnotes, figure numbers, etc.) if such functions were desired. Consequently, each application program developer had to spend time developing programs to implement common printing functions which programs were specific to each application program, thereby resulting in duplicated effort and wasted time.

In order to overcome the aforementioned difficulties, the prior art arrangement was modified as shown in FIG. **2**. In this new arrangement, computer system **200** is still controlled by application program **202** which cooperates, as shown schematically by arrow **204**, with operating system **206**. However, in the system shown in FIG. **2** operating system **206** includes printer drivers **214**. A separate printer driver must still be provided for each different type of printer, but the printer drivers are sold with, and part of, the operating system. Consequently, it is not necessary for each application program to have its own set of printer drivers. An application program, such as application program **202**, communicates with the printer driver **214** by means of a standardized interface **210**. Two common interfaces are called "grafports" or "device contexts". Illustratively, application program **202** provides information (shown schematically shown by arrow **208**) in a standardized form to the grafport **210**. The grafport **210**, in turn, forwards information, as shown by arrow **212**, to printer driver **214** which reformats and converts the information as previously described into the format required by the printer. The output of printer driver **214** is provided (illustratively shown as arrow **216**) to printer port **218** where it is converted to electrical signals that are transmitted, via cable **220**, to the printer **222**.

The configuration shown in FIG. **2** has the advantage that the application program developer need not worry about the specific computer and printer combination on which the program will ultimately run in order to provide printing capabilities to the application program. However, it still suffers from the drawback that, if desired, high level printing capabilities such as pagination and page composition must still be designed into each application program, thereby duplicating program code and wasting programming resources.

SUMMARY OF THE INVENTION

The foregoing problems are overcome and the foregoing object is achieved in an illustrative embodiment of the invention in which an object-oriented printing interface includes document grouping or folio objects which, once instantiated provide complete and flexible printing capability that is transparent to the application program. The printing interface objects include objects that provide query, data transfer, and control methods.

The inventive object-oriented printing interface communicates with the remainder of the operating system by means of a standard interface such as a grafport and printer drivers are provided for each printer type within the operating system. Thus, an application not only need not worry about the particular printer/computer combination with which it is to operate, but also need not have a built in document formatting capability. The printing system includes objects that provide queries for device identification, optimized imaging, and printer status. Other objects are also provided for data transfer to bracket connections prior to sending and receiving information. Still other objects are provided for canceling a print job, pausing a job, and clearing out a job. Finally, an object is also provided for supporting multiple streams of communication to an imaging task.

BRIEF DESCRIPTION OF THE DRAWINGS

The above and further advantages of the invention may be better understood by referring to the following description in conjunction with the accompanying drawings, in which:

FIG. **1** is a schematic block diagram of a prior art computer system showing the relationship of the application program to the operating system and the printer driver in the prior art;

FIG. **2** is a schematic block diagram of a modification of the prior art system shown in FIG. **1** to allow the application program to interface to a standard printing interface in the prior art;

FIG. **3** is a block schematic diagram of a computer system, for example, a personal computer system on which the inventive object-oriented printing interface operates in accordance with a preferred embodiment;

FIG. **4** is a schematic block diagram of modified computer system in which the operating system includes an inventive object-oriented printing interface in accordance with a preferred embodiment;

FIG. **5** is a block schematic diagram of the information paths and hardware by which printable information is channeled through intermediate storage to a print device in accordance with a preferred embodiment;

FIG. **6** shows how the personality document is used by different parts of the printing system in accordance with a preferred embodiment;

FIG. **7** details the printer handler components and their interactions in accordance with a preferred embodiment;

FIG. **8** illustrates the relationship between various printer handler classes in accordance with a preferred embodiment;

FIG. **9** is a block diagram of local printing in accordance with a preferred embodiment;

FIG. **10** is a block diagram of remote imaging in accordance with a preferred embodiment;

FIG. **11** is a block diagram of local imaging in accordance with a preferred embodiment; and

FIG. **12** is a block diagram of local imaging to a networked printer.

DETAILED DESCRIPTION OF THE ILLUSTRATIVE EMBODIMENTS

The invention is preferably practiced in the context of an operating system resident on a personal computer such as the IBM, PS/2, or Apple, Macintosh, computer. A representative hardware environment is depicted in FIG. **3**, which illustrates a typical hardware configuration of a computer **300** in accordance with the subject invention. The computer **300** is controlled by a central processing unit **302**, which may be a conventional microprocessor; a number of other units, all interconnected via a system bus **308**, are provided to accomplish specific tasks. Although a particular computer may only have some of the units illustrated in FIG. **3** or may have additional components not shown, most computers will include at least the units shown.

Specifically, computer **300** shown in FIG. **3** includes a random access memory (RAM) **306** for temporary storage of information, a read only memory (ROM) **304** for permanent storage of the computer's configuration and basic operating commands and an input/output (I/O) adapter **310** for connecting peripheral devices such as a disk unit **313** and printer

314 to the bus 308, via cables 315 and 312, respectively. A user interface adapter 316 is also provided for connecting input devices, such as a keyboard 320, and other known interface devices including mice, speakers and microphones to the bus 308. Visual output is provided by a display adapter 318 which connects the bus 308 to a display device 322 such as a video monitor. The workstation has resident thereon and is controlled and coordinated by operating system software such as the Apple System/7, operating system.

In a preferred embodiment, the invention is implemented in the C++ programming language using object-oriented programming techniques. C++ is a compiled language, that is, programs are written in a human-readable script and this script is then provided to another program called a compiler which generates a machine-readable numeric code that can be loaded into, and directly executed by, a computer. As described below, the C++ language has certain characteristics which allow a software developer to easily use programs written by others while still providing a great deal of control over the reuse of programs to prevent their destruction or improper use. The C++ language is well-known and many articles and texts are available which describe the language in detail. In addition, C++ compilers are commercially available from several vendors including Borland International, Inc. and Microsoft Corporation. Accordingly, for reasons of clarity, the details of the C++ language and the operation of the C++ compiler will not be discussed further in detail herein.

As will be understood by those skilled in the art, Object-Oriented Programming (OOP) techniques involve the definition, creation, use and destruction of "objects". These objects are software entities comprising data elements and routines, or functions, which manipulate the data elements. The data and related functions are treated by the software as an entity and can be created, used and deleted as if they were a single item. Together, the data and functions enable objects to model virtually any real-world entity in terms of its characteristics, which can be represented by the data elements, and its behavior, which can be represented by its data manipulation functions. In this way, objects can model concrete things like people and computers, and they can also model abstract concepts like numbers or geometrical designs.

Objects are defined by creating "classes" which are not objects themselves, but which act as templates that instruct the compiler how to construct the actual object. A class may, for example, specify the number and type of data variables and the steps involved in the functions which manipulate the data. An object is actually created in the program by means of a special function called a constructor which uses the corresponding class definition and additional information, such as arguments provided during object creation, to construct the object. Likewise objects are destroyed by a special function called a destructor. Objects may be used by using their data and invoking their functions.

The principle benefits of object-oriented programming techniques arise out of three basic principles; encapsulation, polymorphism and inheritance. More specifically, objects can be designed to hide, or encapsulate, all, or a portion of, the internal data structure and the internal functions. More particularly, during program design, a program developer can define objects in which all or some of the data variables and all or some of the related functions are considered "private" or for use only by the object itself. Other data or functions can be declared "public" or available for use by other programs. Access to the private variables by other programs can be controlled by defining public functions for an object which access the object's private data. The public functions form a controlled and consistent interface between the private data and the "outside" world. Any attempt to write program code which directly accesses the private variables cause the compiler to generate an error during program compilation which error stops the compilation process and prevents the program from being run.

Polymorphism is a concept which allows objects and functions which have the same overall format, but which work with different data, to function differently in order to produce consistent results. For example, an addition function may be defined as variable A plus variable B (A+B) and this same format can be used whether the A and B are numbers, characters or dollars and cents. However, the actual program code which performs the addition may differ widely depending on the type of variables that comprise A and B. Polymorphism allows three separate function definitions to be written, one for each type of variable (numbers, characters and dollars). After the functions have been defined, a program can later refer to the addition function by its common format (A+B) and, during compilation, the C++ compiler will determine which of the three functions is actually being used by examining the variable types. The compiler will then substitute the proper function code. Polymorphism allows similar functions which produce analogous results to be "grouped" in the program source code to produce a more logical and clear program flow.

The third principle which underlies object-oriented programming is inheritance, which allows program developers to easily reuse pre-existing programs and to avoid creating software from scratch. The principle of inheritance allows a software developer to declare classes (and the objects which are later created from them) as related. Specifically, classes may be designated as subclasses of other base classes. A subclass "inherits" and has access to all of the public functions of its base classes just as if these function appeared in the subclass. Alternatively, a subclass can override some or all of its inherited functions or may modify some or all of its inherited functions merely by defining a new function with the same form (overriding or modification does not alter the function in the base class, but merely modifies the use of the function in the subclass). The creation of a new subclass which has some of the functionality (with selective modification) of another class allows software developers to easily customize existing code to meet their particular needs.

Although object-oriented programming offers significant improvements over other programming concepts, program development still requires significant outlays of time and effort, especially if no pre-existing software programs are available for modification. Consequently, a prior art approach has been to provide a program developer with a set of pre-defined, interconnected classes which create a set of objects and additional miscellaneous routines that are all directed to performing commonly-encountered tasks in a particular environment. Such pre-defined classes and libraries are typically called "application frameworks" and essentially provide a pre-fabricated structure for a working application.

For example, an application framework for a user interface might provide a set of predefined graphic interface objects which create windows, scroll bars, menus, etc. and provide the support and "default" behavior for these graphic interface objects. Since application frameworks are based on object-oriented techniques, the pre-defined classes can be used as base classes and the built-in default behavior can be inherited by developer-defined subclasses and either modified or overridden to allow developers to extend the frame-

work and create customized solution in a particular area of expertise. This object-oriented approach provides a major advantage over traditional programming since the programmer is not changing the original program, but rather extending the capabilities of the original program. In addition, developers are not blindly working through layers of code because the framework provides architectural guidance and modeling and, at the same time, frees the developers to supply specific actions unique to the problem domain.

There are many kinds of application frameworks available, depending on the level of the system involved and the kind of problem to be solved. The types of frameworks range from high-level application frameworks that assist in developing a user interface, to lower-level frameworks that provide basic system software services such as communications, printing, file systems support, graphics, etc. Commercial examples of application frameworks include MacApp (Apple), Bedrock (Symantec), OWL (Borland), NeXT Step App Kit (NEXT), and Smalltalk-80 MVC (ParcPlace).

While the application framework approach utilizes all the principles of encapsulation, polymorphism, and inheritance in the object layer, and is a substantial improvement over other programming techniques, there are difficulties which arise. These difficulties are caused by the fact that it is easy for developers to reuse their own objects, but it is difficult for the developers to use objects generated by other programs. Further, application frameworks generally consist of one or more object "layers" on top of a monolithic operating system and even with the flexibility of the object layer, it is still often necessary to directly interact with the underlying operating system by means of awkward procedural calls.

In the same way that an application framework provides the developer with prefab functionality for an application program, a system framework, such as that included in a preferred embodiment, can provide a prefab functionality for system level services which developers can modify or override to create customized solutions, thereby avoiding the awkward procedural calls necessary with the prior art application frameworks programs. For example, consider a printing framework which could provide the foundation for automated pagination, pre-print processing and page composition of printable information generated by an application program. An application software developer who needed these capabilities would ordinarily have to write specific routines to provide them. To do this with a framework, the developer only needs to supply the characteristics and behavior of the finished output, while the framework provides the actual routines which perform the tasks.

A preferred embodiment takes the concept of frameworks and applies it throughout the entire system, including the application and the operating system. For the commercial or corporate developer, systems integrator, or OEM, this means all of the advantages that have been illustrated for a framework such as MacApp can be leveraged not only at the application level for such things as text and user interfaces, but also at the system level, for services such as printing, graphics, multi-media, file systems, I/O, testing, etc.

FIG. **4** shows a schematic overview of an computer system utilizing the object-oriented printing interface of the present invention. The computer system is shown generally as a dotted box **400**, and an application program **402** and an operating system **406** are provided to control and coordinate the operations of the computer. Application program **402** communicates with operating system **406** as indicated by arrow **404**. However, in accordance with an embodiment of the invention, rather than communicating directly with a standard interface, such as grafport **410**, application program **402** can now communicate with operating system **406** at a higher level when it wishes to print information. This latter interaction is accomplished by providing an object-oriented printing interface shown schematically as box **424**. As will hereinafter be described in detail, printing interface **424** responds to a series of simple commands generated by application program **402** in order to perform various formatting and pagination functions. The formatted, printable information is then transmitted to a grafport **410** as indicated schematically by arrow **426**. It is possible for application program **402** to communicate directly with grafport **410** as in the prior art arrangement, however, it is not contemplated that most applications will do this unless special procedures are needed.

In any case, the information flows through grafport **410**, and as indicated by arrow **412**, is provided to a printer handler **414**. Printer handler **414** is similar to printer drivers previously described. However, it is "intelligent" and offers some additionally capabilities which will be described herein. Essentially, printer handler **414** processes the incoming data stream indicated by arrow **412** and adds the necessary printer commands to control the associated printer schematically illustrated as printer **422**. The reformatted data is provided, as indicated by arrow **416**, to a printer port **418** which converts the data into electrical signals that are sent over cable **420** to printer **422**.

The actual mechanism by which a document generated by the printing interface **424** is transmitted to printer **422** is shown in more detail in FIG. **5**. In particular, printing interface **424** (as will hereafter be described in detail) generates an entity called a document folio shown schematically as document folio **500** in FIG. **5**. The document folio may consist of text, graphics or a combination of the two, all formatted and arranged in a manner specified by the application program. The document folio information is provided, as indicated by arrow **512**, to a print channel **502**. Print channel **502** is an object which is created to transport the information to an appropriate printer. Print channel uses a print job description and a printer identification provided by the application program to transmit the printable information to the appropriate printer.

More specifically, after the print channel **502** receives a printing job, it transmits the printable information to a spooler program **518** as indicated by arrow **514**. Spooler **518** receives the incoming information stream and stores it, in incoming order, in an intermediate storage location **522** as indicated by arrow **520**. Print channel **502** then sends a notification to a print server program **506** via a link **504**, which notification informs print server program **506** that a new print job exists. The print server program **506** is standard program which monitors all print jobs that have been created and also checks the available printers to determine their status.

Once a print job has been completely spooled or stored in intermediate storage **522**, the print server **506** notifies a printer handler **510** by means of a link **508**. The printer handler **510** is type of printer driver which controls and drives a specific printer; its purpose is to convert text and graphic information into printer readable form for any particular printer type. Typically, a printer handler can only process one print job and any other jobs that are created and directed towards the associated printer are queued for later processing. The printer handler contains a despooler program (indicated as box **516**) which retrieves the spooler data from intermediate storage **522** (as indicated by arrow **524**)

and provides the information, as indicated by arrow **526**, to an imaging engine schematically illustrated as box **532**. The imaging engine **532** converts the incoming data stream into the command signals which are necessary to drive the printing elements to produce the final printed document. The commands, indicated schematically by arrow **430**, are provided to the actual print device indicated by box **528** for printing.

Once a print job is completely printed, the printer handler **510** checks its queue for another print job and, if a job is present, begins processing it. Alternatively, if there are no new print jobs to process, the printer handler becomes inactive. After a particular print job is completed, the information stored in intermediate storage in **522** is deleted and the storage is reclaimed. As an option, the storage can retained until a user explicitly deletes the storage.

The printer handler framework facilitates creation of frameworks for different types of printers like PostScript, raster, vector, and PCL. A preferred embodiment provides a framework that is extensible so that new printers can be added to the system very easily. This includes printers with new imaging models, communication protocols, etc. The preferred embodiment also provides a framework that does most of the work to support a printer and at the same time provides enough flexibility for developer customization. The preferred embodiment also allows developer customization at various times during the printing process. Customization occurs at the desktop printer level for presenting device specific user interface, at print time for setting print time features (like duplex printing, multi-bin paper feeding, etc.), at imaging time, by providing a way to access the device at the beginning/end of document and page, and by providing a way to customize rendering of graphics model primitives. Finally, at the device level to support different communication protocol(s).

Clients

All printer manufacturers are clients of the printer handler framework. A client that uses a framework to design PostScript, raster, vector, and PCL printer handler frameworks. Developers start from one of these special types of frameworks and customize it for their printers. For example, a PostScript printer developer (like QMS or Linotype) would use the PostScript printer framework, a plotter developer (like CalComp) would customize the vector printer handler framework.

Architecture

The desktop printer is the only user visible part of the printer. All user actions (commands) related to the printer are handled by this object. When a document is to be printed, it is dragged to the printer icon (or the Print item is selected from the system wide menu). This starts the printing process by first spooling the document by packaging it as a print job. Spooling is a process of streaming the graphic description of a document to the disk so it can be imaged later (possibly in a different task) to the actual printing device represented by the desktop printer. The spooled print job is stored in the printer's persistent data which consists mainly of the print job queue and the printer's dynamic configuration. After the print job is spooled, the printer handler is sent a message that there is a print job for it to process. The printer handler picks up the print job, despools it, converts it to the printer's native imaging model and sends it to the printer.

User's Model of Printing

A reference to a printer in a preferred embodiment really mean a printer model and its associated files, which includes the printer handler. A model is a class that is subclassed from an existing class. Since all user visible entities on a system are models, or the interface presented by them, it makes sense to talk about a printer this way. Each printer model keeps its persistent data (job queue, configuration, etc.) in its model store. The model store provides a way for a model to create separate files that it alone manages. It has protocol for interfacing to a file system. The printer "component" has certain dependencies that must be satisfied when it is moved between machines or enclosed in a business card. A printer is typically dependent on its personality document, the shared library and archive for the system classes that implement the printer, and the shared library and archive for the developer supplied customizations.

When a user "installs" a printer handler in the system, it is immediately available for direct connect devices or in the network browser for network devices. This processing is facilitated by creating a physical model for a direct connect device and a service adapter is "registered" for a network device. A physical device object represents a real device that can be connected directly to the computer (as opposed to available on the network). It is capable of creating a subclass that represents the device. A service adapter indicates the directory protocols (AppleTalk Name Binding Protocol-(NBP), Domain Naming System(DNS) etc.) and service identifiers ("LaserWriter") it can handle and is called on by a directory service to create a model for a service available on a physical directory. To print to a direct connect device, a user connects the printer to the computer (for serial devices) and then drags a document to it. To print to a network device, either the document is dragged to the printer in the network browser or the printer is dragged to the desktop and then the document is dragged to it.

Printer Personality Document

A personality document is supplied by the device manufacturer. The printer personality "document" contains instances of objects, just the data, that implement a particular printer. In other words, it is a shared library without code—just the archive. Examples of objects in the personality document are the printer configuration, print job description which specifies the print time options available on the printer, and the print device object that converts the graphic data into the printer's imaging model. FIG. **6** shows how the personality document is used by different parts of the printing system. The personality document supplied by the developer is used in read-only mode by the printing system. The desktop printer and the printer handler "read" this document to access its personality objects polymorphically.

The analogy of a printer model and its personality to an application and its document implies that a printer model can "read" many personality documents. However, in most cases there is only one personality document per printer because a printer model represents one physical printer. In the case where the user has more than one printer of the same type (for example, two LaserWriter IIg printers), one personality document may be "shared" by multiple printers. The desktop printer obtains user interface objects from the personality (the objects that present the user with printer configuration, features and settings that can be manipulated). The printer handler gets imaging objects from the personality and calls on them to reproduce the document on the printer. Once the printer's dynamic state is added to its persistent store, both the desktop printer and the printer handler refer to it for the printer's configuration. A personality document is created for each type of printer that a

printer handler is created for. The document is created and given to the developer of that type of printer. The developer can "edit" the instance data of objects or replace them with the printer specific implementations.

Printer Handler

FIG. **7** details the printer handler components and their interactions in accordance with a preferred embodiment and FIG. **8** illustrates the logical relationship between various printer handler classes in accordance with a preferred embodiment. The printer handler server team is started by the desktop printer when a print command is initiated by either dragging the document to it or selecting the Print command from one of the menus. The printer handler program creates a dispatch giving it a dispatcher and a message stream to be used as a transport between the client and the server. The dispatch task combines the transport and the dispatcher to create the printer handler server.

There is one printer handler task per physical printer. The printer handler consists of a dispatcher and an imaging task. The dispatcher is a task that handles requests to print jobs and sends them to the imaging task so that the server task is free to handle other requests (more print jobs, status queries, notification, etc.). The printer handler architecture allows for more than one task working on print jobs. Instead of having only one imaging task, the printer handler dispatcher can have a pool of tasks that access the job queue and pick up a job to process. This way, the printer handler can be processing multiple jobs at the same time. If the target printer can accept only one job at a time, only one imaging task will be talking to it at a given time. The multiple imaging tasks model works well when the destination is a spooler that can accept more than one job at a time. Each job in the queue will know the task that is processing it so things like abort, pause, etc. can function properly.

Printer Handler Server

The printer handler task is started by the desktop printer when a print command is initiated by either dragging the document to the printer icon or selecting the Print command from one of the menus. The desktop printer has a client class that starts up the server. The client class provides the protocol for calling "remote" procedures that are implemented in other objects, tasks, or servers. The printer handler program creates a transport and a dispatcher to create the printer handler server.

There is one printer handler task per physical printer. The printer handler consists of a dispatcher and an imaging task. The dispatcher handles requests to print jobs and sends them to the imaging task so that the server task is free to handle other requests (more print jobs, status queries, notification, etc.). The printer handler architecture allows for more than one task working on print jobs. Instead of having one imaging task, the printer handler dispatcher has a pool of tasks that access the job queue and pick a job to process. This way, the printer handler can be processing multiple jobs at the same time. If the target printer can accept only one job at a time, only one imaging task communicates to it at a given time. The multiple imaging tasks function efficiently when the destination is a spooler that can accept more than one job at a time. Each job in the queue understands the task that is processing it so things like abort, pause, etc. are managed properly.

Printer Handler Imaging Task

The printer handler imaging task receives one job at a time. It uses the developer customizable imaging objects to convert the source description of the document into a stream of data targeted for a particular printer. The imaging task obtains a spool stream from the print job and passes it to the a printer device object. The printer device object extracts individual pages out of the print run and converts the individual pages into the printer's imaging model. The imaging objects also perform the task of mapping the attributes requested by the print job (page size, media, color, fonts, etc.) to features actually available on the printer. This processing is achieved by consulting the printer's dynamic state maintained by the printer handler.

The imaging objects produce an output stream that is sent to the output device. The framework for a specific type of printer defines an appropriate class. For example, the PostScript printer handler framework defines a class whose subclass talks with a printer using the AppleTalk Printer Access Protocol (PAP). A TCP/IP based printer can be easily supported by subclassing the same class.

Printer's Persistent Data

The printer handler is responsible for keeping track of the printer's persistent data, in particular, its job queue and dynamic state. The dynamic state contains things like the current media/tray setting, current color mode (2, 3 or 4 colors), etc. Since each printer would want to save different things in its dynamic state, there is a class that developers can subclass to add printer specific features. For the most part, it is this class that will be streamed out to the printer's persistent data store. The default implementation of the persistent data store will be a dictionary, although the developer is free to use some other storage scheme. Since the printer's state is persistent, the printer handler can be easily restarted in case of a crash or system shutdown.

The printer state, which is part of the printer model's store, is updated when the printer handler images a job to it and finds that the state has changed. This scheme works when the printer handler has two-way communication available with the printer. In cases when two-way communication is not available, the printer handler will rely on the printer's static configuration.

Status Reporting

One of the goals of the printer handler framework is to provide a convenient way for developers to report status and notification from the printer to the user. The printer handler framework employs the same facilities provided by standard frameworks. It is standard procedure to report normal progress information to the user as a print job progresses. There are two types of statuses that a printer handler might want to report. The first is the global status of the job, for example, "Processing page 3 of 50", or "Printing 4th out of 10 copies", etc. This type of global job status is common for all printers and can be provided easily by the framework. The second kind of status is one that comes directly from the printer, for example, "user: Jay Patel; job: Printer Handler ERS; status: busy; source: AppleTalk". Some printers may not report this type of status at all.

User notification is given in cases where there is a problem with printing. This may be a result of a printer out of paper, a paper jam, communication error, PostScript error, a plotter needs new/different set of pens, printer is out of ribbon, etc. For some of these situations, the user must be notified and the printing process can continue once the problem is rectified. There are cases, however, where the printer may not be able to say that the problem is fixed. In such cases, a notification must be given to the user and the printing process must wait until the user says it's OK to continue.

Printer Handler Classes

A discussion of the classes that implement the printer handler framework is provided below.

TPrinterHandler

```
class TPrinterHandler : public MRemoteDispatcher {
    public:
        TPrinterHandlero( );
        virtual-TPrinterHandler( );
    private:
    // Server Requests
    // Every XXXRequest method unflattens arguments, if any, and then
    // calls the corresponding HandleXXX method. It then calls
    // ReturnSuccess and flattens results to the result stream.
        // Job Queue Management
        // Requests that apply to all jobs in the queue
            void        GetAllJobsRequest( );
            void        AbortAllJobsRequest( ):
            void        DeferAllJobsRequest( );
            void        UndeferAllJobsRequest( );
            void        DeferAllJobsUntilRequest( );
        // Requests that apply to one job in the queue
            void        AddJobRequest( );
        // RemoveJobRequest will abort the job if it is currently
being processed.
        // Otherwise, the job will be removed from the queue.
            void        RemoveJobRequest( );
            void        PauseJobRequest( );
            void        DeferJobRequest( );
            void        UndeferJobRequest( );
        // Update printer's state
            void        UpdateDynamicPrinterDataRequest( );
        // return status of a job
            void        GetStatusOfJobRequest( );
        //
        // Subclasses can override the following HandleXXX methods.
        // HandleXXX are called from the corresponding request
methods.
        //
        // Job Queue Management
        // Requests that apply to all jobs in the queue
        virtual void    HandleGetAllJobs( );
        virtual void    HandleAbortAllJobs( );
        virtual void    HandleDeferAllJobs( );
        virtual void    HandleUndeferAlljobs( );
        virtual void    HandleDeferAllJobsUntil( );
        // Requests that apply to one job in the queue
        virtual void    HandleAddJob( );
        // RemoveJobRequest will abort the job if it is currently
being processed.
        // Otherwise, the job will be removed from the queue.
        virtual void    HandleRemoveJob( );
        virtual void    HandlePauseJob( );
        virtual void    HandleDeferJob( );
        virtual void    HandleUndeferJob( );
        // Update printer's state
        virtual void
    HandleUpdateDynamicPrinterData( );
        // return status of a job
        virtual void    HandleGetStatusOfJob( );
        // for TPrinterHandlerCaller only
        typedef enum {
            kGetAllJobsRequest, kAbortAllJobsRequest,
kDeferAllJobsRequest,
            kUndeferAllJobsRequest, kDeferAllJObsUntilRequest,
            kAddJobRequest, kRemoveJobRequest, kPauseJobRequest,
            kDeferJobRequest, kUndeferJobRequest,
            kUpdateDynamicPrinterDataRequest,
            kGetStatueRequest
        };
        friend class TPrinterHandlerCaller; // so it can use enums
above.
    protected:
        // Get the imaging task to send jobs to
        virtual TPrinterImagingTask* GetImagingTask( );
        // Get the job queue for this printer
        virtual TDeque* GetPrintJobQueue( );
```

-continued

TPrinterHandler

```
        // . . . . . Methods to communicate with the imaging task
        // . . . . . .
};
```

TPrinterHiandler is a base class that provides protocol for dispatching server requests. The corresponding client class TPrinterHandlerCaller is described later. TPrinterHandler maintains a job queue for the printer. This queue is semaphore protected to allow concurrent access by the printer handler and an imaging task. GetImagingTask creates a TPrinterImagingTask giving it an exchange to communicate with (an exchange provides a place to send messages to and receive messages from). TPrinterImagingTask is given one job at a time to process (by AddJobRequest). When the job is finished, it notifies the printer handler so it can decide what to do with the job.

GetStatusOfJobRequest returns status of a job in the job queue. For the job being processed currently, the status reported is the "global" job status described earlier. There are two ways this could be implemented. One way is for the TPrintDevice subclass to post the status (perhaps a TText) periodically at a global location which the printer handler returns to the client in GetStatusRequest method. Another way is to implement a helper task to get status from the TPrintDevice subclass. For any other job (not currently being processed), the status that is returned might be the number of pages in the job (if that's appropriate), how far down the queue this job is, etc.

TPrinterHandlerCaller

```
    class TPrinterHandlerCaller : protected MRemoteCaller {
    public:
        TPrinterHandlerCaller(TSenderTransport*);
        virtual-TPrinterHandlerCaller( );
        // Remote requests
        // These are called by TPrinterModel's command handlers
        virtual TPrintJobQueue*     CreateJobIterator( );
        virtual void                AbortAllJobs( );
        virtual void                DeferAllJobs( );
        virtual void                UndeferAllJobs( );
        virtual void                DeferAllJobsUntil( );
        virtual void                AddJob(TPrintJobSurrogate&);
        virtual void                RemoveJob(const
TPrintJobSurrogate&);
        virtual void                PauseJob(const
TPrintJobSurrogate&);
        virtual void                DeferJobRequest(const
TPrintJobSurrogate&);
        virtual void                UndeferJobRequest(const
TPrintJobSurrogate&);
        virtual void                GetStatus(TText&);
        // . . . . .
    MRemoteCallerDeclarationsMacro(TPrinterHandlerCaller);
};
```

A TPrinterHandlerCaller is instantiated in the printing task (the task that initiates printing, probably a compound document) by the TPrinterModel. It uses a transport to send a request to the printer handler task. The transport can be local or remotely located. Thus, the printer handler to be on a remote machine. A reference to an already-registered service (like a network printer) required by the transport is known to the printer that the document is being printed on. When the printer handler is remote, TServiceReference is obtained from the network.

TPrinterHandlerCaller's methods are called by the printer model's commands which are called by the document framework in response to user actions.

TPrinterImagingTask

```
class TPrinterImagingTask : public TTaskProgram {
    public:
        TPrinterImagingTask(TIPCExchange*);
        virtual     ~TPrinterImagingTask( );
        // TTaskProgram override
        virtual void     Run( );
        // Support methods to handle different types of messages
from the
        // Printer Handler.
        // The messages that this task will receive are:
        // AbortJob
        // PauseJob
        // GetStatus
        // ProcessJob
        // etc.
        // . . . . . .
};
```

TPrinterImagingTask, which is created by the printer handler, performs the task of imaging print jobs and supplying progress information for the same. The constructor receives a TIPCExchange that the imaging task uses to communicate with the printer handler. As far as the communication between the printer handler server and the imaging task is concerned, there are two possibilities. One is to use the exchange to receive messages and dispatch them based on the message id that the printer handler attaches to the header. Another way to do Inter-Process Communication (IPC) is to use wait groups to handle the dispatching automatically when you provide message handlers for each type of message. The second method makes the implementation more structured (avoids a switch statement) but involves writing more code.

ProcessJob gets a TPrintJobSurrogate as a parameter. Using the TPrintJobSurrogate, the imaging task gets to the print job. A print job has a reference to the printer's persistent data (a TDiskDictionary) that keeps the spooled image of the document, a reference to the printer that the job was targeted to, etc. The printer reference (lets call it TPrinterIdentity) is actually a reference to the TPrinterModel's data. Using this data the imaging task can get to the TPrintDevice subclass for the printer. The imaging task gets the spool stream and the job description (TPrintJobDescription) out of the job's persistent data and asks the print device to process it.

TPrintDevice

```
class TPrintDevice : public MCollectible {
    public
        virtual ~TPrintDevice( );
// Don't override these: override the HandleXXX methods below.
    virtual void RenderPrintRun(const TPrintRun&,const
TPrintJobDescription&.
                const TPrinterIdentity&);
        virtual void RenderPage(const TGrafRun& grafRun, const
TPageDescription&,
// MCollectible support
        virtual TStream& operator<<=(TStream& fromwhere);
        virtual TStream&operator>>=(TStream& towhere) const;
        virtual Boolean IsEqual(const MCollectible *) const;
    protected;
```

-continued

TPrintDevice

```
        TPrintDevice& operator=(const TPrintDevice&);
        // You can use these in the HandleXXX methods below to
get current
        // page/job information.
        virtual TPrintRun* GetPrintRun( );
        virtual const TPrintJobDescription*
GetPrintJobDescription( );
        virtual TGrafRun* GetGrafRun( );
        virtual TPageDescription* GetPageDescription( );
        // The following methods are called as a result of
RenderPrintRun( ).
        // Don't call these directly: call RenderPrintRun( ).
        // You may override these. If you override
Begin/EndPrintRun, then
        // be sure to call these base class methods as the first
thing
        // in your override implementations.
        virtual void     HandleBeginPrintRun( );
        virtual void     HandleRenderPrintRun( );
            // Default implementation goes through the
printrun in
            // forward order and calls RenderPage for each
page.
        virtual void     HandleEndPrintRun( );
        // The following methods are called as a result of
RenderPage( ).
        // Don't call these directly: call RenderPage( ).
        virtual void     HandleBeginPage( );
        virtual void     HandleRenderPage( ) = 0;
        // You must override this to convert the page to the
printer imaging model.
        virtual void     HandleEndPage( );
    protected:
        TPrintDevice( );
};
```

TPrintDevice converts a document to the printer imaging model. It provides an abstract interface to access page and job information and to process the job (a Print Run) and each page. Subclasses implement HandleRenderPage method to convert the page data to the printer imaging model. TPrintDevice is one of the objects that will be included in the personality document that the developer can edit or subclass. Therefore, it is possible for the developer to supply an implementation for converting the document to the printer imaging model. TPrintDevice is resurrected from the printer personality by the imaging task. RenderPrintRun is called with a TPrintRun, TPrintJobDescription, and a TPrinterIdentity. RenderPrintRun calls HandleBeginPrintRun. HandleRenderPrintRun, and HandleEndPrintRun. The reason for providing the begin and end methods is so that the subclasses can send some global commands to the printer before and after the document is processed. The default implementation of HandlePrintRun goes through the printrun in forward order and calls RenderPage for each page. Subclasses can override this to "play" the document in any random order they like. RenderPage calls HandleBeginPage, HandleRenderPage and HandleEndPage. Again, the reason for providing the begin and end methods is so subclasses can send page level global commands to the printer.

The TPrintJobDescription parameter, passed in the constructor, gives the user selected print time options. The print device maps the user's choice to what is actually available on the printer. It uses the printer's configuration kept in the printer's persistent store (the printer identity object can be used to get to the persistent store.).

TPrinterConfiguration

```
class TPrinterConfiguration : public MCollectible {
    public:
        virtual-TPrinterConfiguration( );
        // for static state of the printer
        // virtual TPageDescription&
    GetDefaultPageDescription( ) const = 0;
        // subclasses can return static or current state of the
printer from    // following methods.
        virtual TPageList&          GetPageList( ) const =
0;
        virtual TMediaList&         GedMediaList( ) const =
0;
        virtual TResolutionList&    GetResolutionList( )
const = 0;
        virtual TRGBColor           GetEngineColor( ) const =
0;
        // for dynamic state of the printer
        virtual TPageDescription&
    GetCurrentPageDesecription( ) const = 0;
        virtual void                SetPageList(TPageList&) =
0;
        virtual void
    setMediaList(TMediaList&) = 0;
        virtual void
    SetResolution(TResolution&) = 0;
        virtual TStream&      operator<<=(TStream& fromwhere);
        virtual TStream&      operator>>=(TStream& towhere)
const;
        virtual Boolean       IsEqual (const MCollectible*) const;
        virtual long              Hash( ) const:
    protected:
        TPrinterConfiguration( );
        TPrinterConfiguration&operator=(const
TPrinterConfiguration&);
};
```

TPrinterConfiguration is an abstract base class for a printer's state. It is part of the printer's persistent data kept current by the printer handler. Subclasses can store the actual configuration data in a file of their choice (e.g. dictionary, flat stream, etc.). For example, TPSPrinterConfiguration will use PPD files converted to a disk dictionary to keep configuration data. TPrinterConfiguration defines a protocol that provides for setting and getting configuration items such as page sizes, media, resolution(s), etc. When a printer is first available for use, its persistent store (a TModelStore) is created and TPrinterConfiguration is asked to write itself into the store. This becomes the printer's initial state which is updated when a print job is sent to it.

The lists returned by getters (TPageList, TMediaList, etc.) are implemented using C++ templates. As mentioned earlier, each type of printer has a subclass of TPrinterConfiguration that returns the printer's static configuration. This is streamed into the personality document which is given to the developer of that type of printer. The developer typically will edit the configuration instance (that is, the fields of the particular TPrinterConfiguration class) to include the printer's data.

TPrintJobDescription

```
class TPrintJobDescription : public MCollectible
{
    public:
        MCollectibleDeclarationsMacro(TPrintJobDescription);
    public:
    TPrintJobDescription(TPrinterIdentity&);
```

-continued

TPrintJobDescription

```
        TPrintJobDescription(const
        TPrintJobDescription&);
    virtual ~TPrintJobDescription( );
    typedef enum EBannerPage { eNoBanner, eBriefBanner,
eWholeBanner
};
        // -------------------------          //
Description: These member functions are pretty much self
explanatory
        //                  except for notes as added.
        // Rewires          :
        // Modifies         : The job's state is altered to reflect the
requested
        //              operation.
        // Effects          :
        // Raises           :
        // Override         : All subclasses must override all virtual
functions.
        // ----------------------------                          virtual
void        SetCopies(unsigned long);
        virtual unsigned long       GetCopies( ) const;
        virtual unsigned long       GetPageCount( ) const;
        virtual void                SetPageCount(unsigned long);
        virtual void                SetCoverPage(EBannerPage);
        virtual EBannerPage         GetCoverPage( ) const;
        virtual void                SetEndPage(EBannerPage);
        virtual EBannerPage         GetEndPage( ) const;
        // ----------------------------                          // Punt
choice specifies what the user wants to do when the page sizes
    // don't match between the document and the printer.
These options
        // are defined in PageDescription.h
        // ----------------------------                          virtual
void            SetJobPuntChoice(EPuntOption choice);
        virtual EPuntOption         GetJobPuntChoice( ) const;
            // Get the printer that this PrintJobDescription
comes from
        virtual void
    GetPrinterIdentity(TPrinterIdentity&) const;
        // User interface. Equivalent to the classic Print Job
dialog.
        virtual void                EditSelf( );
        // MCollectible support
        virtual TStream&            operator<<=(TStream&
fromwhere);
        virtual TStream&            operator>>=(TStream&
towhere) const;
        virtual TPrintJobDescription& operator=(const
TPrintJobDescription&);
        virtual Boolean         operator==(const
TPrintJobDescription&) const;
        virtual Boolean             IsEqual(const MCollectible*
right) const;
    protected:
            TPrintJobDescription( ); // for Streaming
};
```

TPrintJobDescription is a base class providing protocol for accessing/changing print time options like number of copies, cover page choices, punt options (what to do when there is a mismatch between the document and printer page sizes), etc. Developers can subclass this to add features specific to their device. The default implementation of TPrintJobDescription provides the choice of options common to all printers. The print job description gets streamed with a print job and is used by TPrintDevice (in the printer handler task) to send appropriate commands to the printer that implement the print time options. Each job description knows the printer that it is comes from. As a matter of fact, it is created by the printer model. The printer model gets it out of the printer's personality document. EditSelf is a place holder for a method that might be used to allow users to change job description attributes. This method might be called in response to the system wide print menu command.

User Scenario

A new PostScript printer, ABC Lriter is added to the network that the user wants to print to. Since ABC chose not to subclass any of the personality classes, it simply supplies the PPD (Postscript Printer Description) file with the printer. The TPSPrinterConfiguration can parse a PPD file and convert it into a configuration dictionary. So, the PPD file can be placed somewhere on the computer, and a preferred embodiment handles the rest. The ABC Lriter Iicon is moved from the network browser to the printer in the browser. Behind the scenes, the TPSPrinterServiceAdaptor class (a subclass of TServiceAdaptor), given a TServiceReference in the CreateModel method, talks to the printer to get the product name of the printer and associates the model with the correct PPD file. If the printer is busy, this operation may have to be deferred. Also, if the device specific PPD file is not available. TPSPrinterConfiguration will default to a "standard" PostScript printer configuration.

Printer Sharing

Sharing printers across machines (peer to peer or print servers) is supported by the Printer Handler. Compatibility with existing OS applications, legacy applications is a part of a preferred embodiment. Applications and operating systems that abstract the imaging into graphical data can write translation drivers to convert the data format. This has the benefit of being device independent when entering into the system. An example of this kind of translation is a Windows application, layered above Pink, in which the GDI printing driver would translate the GDI into a compatible format. At this point, the print job would be similar to other print jobs. If the application does the imaging, the device independence is lost. The adapter must present specific printer types to the user for selection. The system accepts the raw printer data and pass it onto the printer. For a uniform interface to print jobs, this job should show up in the printer queues as any other job. Due to the lack of direct device manipulation by the application, the imaged data stream will have to make assumptions about the device and will not be redirectable to other output devices.

For legacy data, the approach will be slightly different. If the data can be converted into a system graphic model, then a TModel data encapsulator should be written to do this conversion. Almost any file format, such as TIFF, Pict, HPG L, Adobe Illustrator and Adobe Acrobat should be easily translatable. More complex files such as EPSF or Raw PostScript files can be encapsulated but would require a PostScript interpreter. To simply print these complex files, the same imaged data interface specified above could be utilized. A print job either contains a spool file or imaged data. This is to prevent the mixing of device dependent and device independent data.

Peer-to-peer printing

The ability to print across machines has usually been accomplished by dedicating a server for this purpose. Now that the operating system can accommodate additional remote processing (due in part to protected address spaces, robustness and computation power), the use of a client machine (peer) as a server for other peers is achievable. To allow this, the printing architecture must permit printer handlers to advertise themselves as a sharable printer and the printer handler must be able to transfer the job to the remote system. The use of a peer as a server clouds the distinction but for our purposes all machines will have the same capabilities. The printing system will not treat systems setup for serving multiple printers differently than a published peer.

Due to the raw data requirement, the printing system must be able to spool both graphics between systems and raw imaged data. The case of graphics is considered remote imaging because the device independent data is converted to device dependent data on the remote system. Conversely, the case of spooling raw data can be thought of as local imaging since you could actually convert the data into an imaging language on the local system before transferring the device dependent data to the remote printing system. This capability is only available due to the legacy data requirement. The use of the local imaging model constrains the print job by taking away the device independent nature of the spool file.

Printing Scenarios

The following printing scenarios will help illustrate the use of the shared printer handler. Some of the scenarios are similar but are presented for thoroughness.

Local printing

FIG. **9** is a block diagram of local printing in accordance with a preferred embodiment. Although local printing does not involve printer sharing, it is useful to consider this as a base model for future discussion. To print to a local printer, the document makes a print request through Tprinter::AddJob( ). This translates into a TPrinterHandlerCaller which is a subclassed MRemoteCaller. The corresponding TPrinterHandler (subclassed from MRemoteDispatcher) will accept the print job and enqueue the job for printing. The Printer Handler will schedule the job through the TImagingTask and the TPrintDeviceChannel. The corresponding TPrintDeviceChannel provides the communication (imaged data transfer, command and status) to the local printer. If the data is legacy imaged data, the TImagingTask does not have to translate the spool file but instead passes it on to the TPrintDeviceChannel without interpretation (ie., it literally dumps the bytes).

Remote imaging

Remote imaging consists of the local application connecting with a local Printer Handler which spools the graphics commands onto the local system. The Printer Handler connects to a remote system through the same TRemoteCaller/TRemoteDispatcher mechanism as the local print jobs. The remote Printer Handler then routes the spool job through the imaging task and out to the printer via a TPrintDeviceChannel. By routing the job through the remote Printer Handler, the print job will be prioritized in the remote queue. FIG. **10** is a block diagram of remote imaging in accordance with a preferred embodiment.

Local imaging

In a local imaging scenario, the application spools the graphics to the local disk but instead of sending the spool file to a remote system, the graphics are translated into an imaging language (like PostScript) by the local imaging task. The local Printer Handler then sends the imaged data to the remote system (the remote Printer Handler) for spooling into its spool queue. The job is then sent out to the printer via a local TPrintDeviceChannel. The ability to spool raw imaged data is utilized for this situation but should be avoided since the device independence is lost prior to being received by the remote system. The use of device queries or bi-directional communication with the device is not possible across remote systems. FIG. **11** is a block diagram of local imaging in accordance with a preferred embodiment.

Local imaging to networked printer

Networked printers are basically the same as a local printer except that the communications channel must use a specific network protocol. This scenario can be used for printers using protocols based on NPA (Network Printing Architecture), TCP/IP, PAP protocol, and Novell IPX based print servers. The TPrintDeviceChannel will map the printer protocol into the TImagingTask interface for devices. FIG. **12** is a block diagram of local imaging to a networked printer.

TPrinterHandler Interface Changes

A TPrinterItandler will register itself as a TServiceAdaptor and provide a protocol interface (using Message Streams. Since the communication is similar whether the TPrinterHandler is on one machine or multiple machines, it is not necessary to implement a wrapper class to handle the IPC between machines.

TPrintlobHandle Interface

To facilitate printing raw data from a TModel, the TPrintJobHandle interface will include a method to set the type of the job (SetJobType).

TPrintDeviceChannel Interface

The TPrintDeviceChannel interface will be used to provide a common interface between the imaging task and the physical printer. The TPrintDeviceChannel will implement different communications protocols for the printing services, example implementations could be for direct connect printers (serial or parallel) or networked devices (such as PAP, Novell, and TCP/IP). For some hardware or special purposes, the TPrinterHandler can integrate the communications portion without using a TPrintDeviceChannel but typical printer architecture's will use TPrintDeviceChannel.

In summary, the printer handler interface provides a set of query, data transfer and control methods. The queries consist of a Lookup (for device identification), IsBidirectional (in order to provide optimized imaging), and GetStatus (for printer status). For data transfer, the Connect and Disconnect methods will be used to bracket connections prior to using SendData and GetData. The control methods include Abort (cancel the job), Pause (pause the job) and Flush (to clear out the data channel). To allow for parallel processing between the imaging task and the TPrintDeviceChannel, a GetStream method provides multiple streams to the imaging task.

While the invention is described in terms of preferred embodiments in a specific system environment, those skilled in the art will recognize that the invention can be practiced, with modification, in other and different hardware and software environments within the spirit and scope of the appended claims.

Having thus described our invention, what we claim as new, and desire to secure by Letters Patent is:

1. A computer system for controlling a print device to generate a printed output, the computer system comprising:

(a) an application program for generating printable information;

(b) a storage device;

(c) a processor for executing the application program, for storing the printable information in the storage device and for retrieving the printable information from the storage device;

(d) an operating system stored in the storage device and cooperating with the processor for controlling the print device, the operating system comprising:

(1) a printing interface for formatting and paginating the printable information in response to commands generated by the application program;

(2) a printer handler for receiving formatted printable information from the printing interface, for converting the formatted printable information to a native imaging model of the print device, for modifying the printable information to include predetermined printer commands to control the print device and for providing an output data stream which includes the printable information modified to include the predetermined printer commands; and

(3) means for transmitting the formatted, printable information from the printing interface to the printer handler; and

(e) communication means for receiving the output data stream from the printer handler and for transmitting the output data stream including the modified printable information to the print device.

2. A computer system as recited in claim **1**, wherein:

the communication means converts the modified printable information data into electrical signals that are sent to the print device; and

the printer handler includes a plurality of query methods a plurality of data transfer methods and a plurality of control methods and wherein at least one of the plurality of query methods is a method for querying the print device for status information.

3. A computer system as recited in claim **1**, wherein the printing interface generates a document folio which includes textual and graphics data which has been formatted and arranged in a manner specified by the application program; and wherein the computer system further includes a print channel object created to transport the document folio to the print device, wherein the application program identifies the printable information with a print job description and a printer identification which the print channel object uses to transmit the printable information to the print device.

4. A computer system as recited in claim **3**, further comprising:

a spooler for receiving printable information from the print channel object;

an intermediate storage location, coupled to the spooler, wherein the spooler stores the printable information received from the print channel object in the intermediate storage location in a predetermined order; and

means for notifying the printer handler that printable information is stored in the intermediate storage location.

5. A computer system as recited in claim **4**, wherein the printer handler further comprises:

a despooler for retrieving the printable information from the intermediate storage location and forming a data stream; and

an imaging engine, coupled to the despooler for receiving the data stream, the imaging engine converting the data stream fed thereto into command signals for driving the print device to provide the printed output.

6. A computer system as recited in claim **1**, including menu means for initiating a print operation.

7. A computer system as recited in claim **1**, including means for connecting the printer handler to a remote computer system having a second print device.

8. A method for controlling a computer to generate a printed output, the method comprising the steps of:

(a) generating printable information comprising at least one of textual data and graphics data using an application program running in a local computer;

(b) formatting the printable information in response to commands generated by the application program;

(c) paginating the printable information in response to commands generated by the application program;

(d) generating a document folio which includes the printable information which has been formatted and paginated by the application program;

(e) spooling the printable information onto a storage device of the local computer;

(f) connecting the local computer to a remote computer system having a print device coupled there to;

(g) converting the printable information to a native imaging model of the print device coupled to the remote computer system;

(h) modifying the printable information to include predetermined printer commands to control the print device of the remote computer system; and

(i) transmitting the formatted, modified printable information to the print device of the remote computer system for printing.

9. A method as recited in claim **8**, including the step of querying the print device of the remote computer system for status information.

10. A method as recited in claim **8**, wherein step (i) includes the steps of:

(i1) storing a graphic description of the documentation folio on the storage device as a print job;

(i2) transmitting a message to a printer handler that there is a print job for the print handler to process;

(i3) locating the print job via the printer handler;

(i4) retrieving the print job from the storage device;

(i5) converting the print job to a native imaging model of the print device; and

(i6) sending the converted print job to the print device.

11. A method as recited in claim **10**, further comprising the step of:

(j) prioritizing the print job in a print queue of the remote computer system.

12. A method as recited in claim **10**, further comprising the steps of:

(k) converting the print job retrieved from the storage device into command signals;

(l) providing the command signals to the print device; and

(m) using the command signals to drive printing elements of the print device and to produce a printed document.

13. A method as recited in claim **12** wherein after a particular print job is completed, the method further includes the step of:

(n) deleting the print job stored in the storage device.

14. A method as recited in claim **8**, including the step of:

(p) reporting status information during a print operation.

15. A method as recited in claim **14**, including the step of:

(q) reporting status information when a new page is encountered.

16. A method of transmitting a document to a printer under control of an application program, the method comprising the steps of:

(a) generating a document folio containing textual and graphic data to be printed, the data being formatted and arranged in a predetermined manner specified by the application program;

(b) generating a print job description and a printer identification code for the printer;

(c) instantiating a print channel object using the printer identification code to transport the document folio to the printer;

(d) providing the document folio and the print job description to the printer channel object as a print job;

(e) transporting the print job to the printer via the print channel object; and

(f) removing the document folio from the print job and using the information therein to control the printer to print the textual and graphic data therein.

17. The method of claim **16** wherein the step (e) includes the steps of:

(e.1) transmitting the print job to a spooler program;

(e.2) receiving the print job in the spooler program;

(e.3) storing the print job in an intermediate storage location; and

(e.4) notifying a print server via a link that the print job is stored in the intermediate storage location.

18. The method of claim **17** wherein, step (e) further comprises the steps of:

(e.5) notifying a printer handler via the print server that the textual and graphic data has been stored in the intermediate storage location;

(e.6) retrieving the textual and graphic data from the intermediate storage location; and

(e.7) providing the textual and graphic data to an imaging engine.

19. The method of claim **18** wherein step (f) further comprises the steps of:

(f.1) converting, via the imaging engine, the textual and graphic data retrieved from the intermediate storage location into command signals;

(f.2) providing the command signals to the printer; and

(f.3) driving printing elements of the printer via the command signals to provide a printed document.

20. The method of claim **18** further including the step of:

(g) deleting the textual and graphic data from the intermediate storage location.

21. A method for enabling printing by a computer having a memory and an operating system, comprising the steps of:

(a) providing class libraries for storage in the computer memory from which

(1) a printing interface object may be instantiated to cause the conversion of application program output to page-formatted text and graphic data, and

(2) a print channel object may be instantiated to cause the transfer of the page formatted data to an appropriate printer; and

(b) providing a run-time environment to

(1) support the instantiation of the printing interface and print channel objects and

(2) selectively cause the transfer by the operating system of the page formatted data to a specific printer.

22. The method of claim 21, wherein the transfer causing step (b)(2) further comprises the step of rearranging the page order of the formatted data responsive to a message representing a print job priority.

23. The method of claim 22, wherein the rearranging step further comprises the step of selectively reorienting the page formatted data responsive to a message representing a predetermined bookler print format.

24. The method of claim 21, wherein the transfer causing step (b)(2) further comprises the step of transferring selected portions of the page formatted data to a specific printer.

25. The method of claim 21, wherein the transfer causing step (b)(2) further comprises the step of transferring the page formatted data to a selected one of several specific printers.

26. The method of claim 21, wherein the page formatted data includes text and graphics data arranged in a predetermined printer-dependent format.

27. The method of claim 21, wherein the application program output includes text and graphics data arranged in a printer-independent format.

28. A computer program product for enabling printing by a computer having a memory and an operating system, said computer program product including a computer-useable means for storing therein computer-readable code comprising:

program code for providing class libraries for storage in the computer memory from which

a printing interface object may be instantiated to cause the conversion of application program output to page-formatted text and graphic data, and

a print channel object may be instantiated to cause the transfer of the page formatted data to an appropriate printer; and

program code for providing a run-time environment to support the instantiation of the printing interface and print channel objects and

selectively cause the transfer by the operating system of the page formatted data to a specific printer.

29. The computer program product of claim 28, further comprising program code for rearranging the page order of the formatted data responsive to a message representing a print job priority.

30. The computer program product of claim 29, further comprising program code for selectively reorienting the page formatted data responsive to a message representing a predetermined booklet print format.

31. The computer program product of claim 28, further comprising program code for transferring selected portions of the page formatted data to a specific printer.

32. The computer program product of claim 28, wherein the page formatted data includes text and graphics data arranged in a predetermined printer-independent format.

33. The computer program product of claim 28, wherein the application program output includes text and graphics data arranged in a printer-independent format.

* * * * *

US 20030023590A1

(19) **United States**

(12) **Patent Application Publication** (10) **Pub. No.: US 2003/0023590 A1**

Atkin (43) **Pub. Date: Jan. 30, 2003**

(54) **GENERALIZED MECHANISM FOR UNICODE METADATA**

(75) Inventor: **Steven Edward Atkin**, Palm Bay, FL (US)

Correspondence Address:
Robert H. Frantz
P.O. Box 23324
Oklahoma City, OK 73123-2334 (US)

(73) Assignee: **International Business Machines Corporation**, Armonk, NY

(21) Appl. No.: **09/838,376**

(22) Filed: **Apr. 19, 2001**

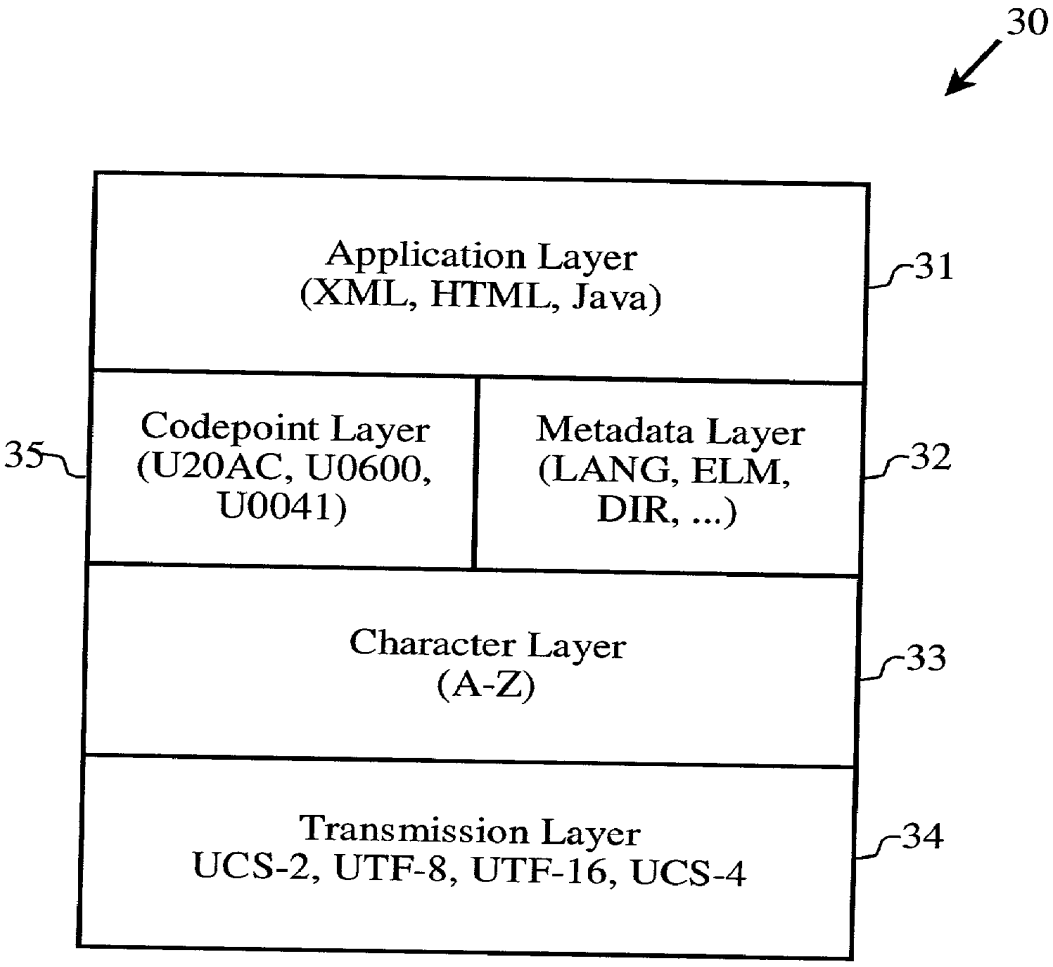
Publication Classification

(51) **Int. Cl.**[7] **G06F 7/00**

(52) **U.S. Cl.** **707/6**

(57) **ABSTRACT**

A extendable method for including display rendering metadata within Unicode character streams. Metadata is distinct from character data, even though it is embedded in the Unicode character stream using tag mechanism. The method allows for an unlimited number of tag identifiers. Legacy Unicode methods such as Bidi, Normalization, and Line Breaking, can be recast using the invention in a more manageable context according to the metadata framework, thereby allowing the methods to be detectable, reversible as well as convertible. The traditional Unicode Control Layer is eliminated because the syntax of controls are captured universally by the new Metadata Layer, irrespective of whether the control relates to presentation or pcontent. By replacing the indistinct boundary separating characters and control with a well defined division, applications that rely on Unicode are easier to develop and to maintain.



Prior Art

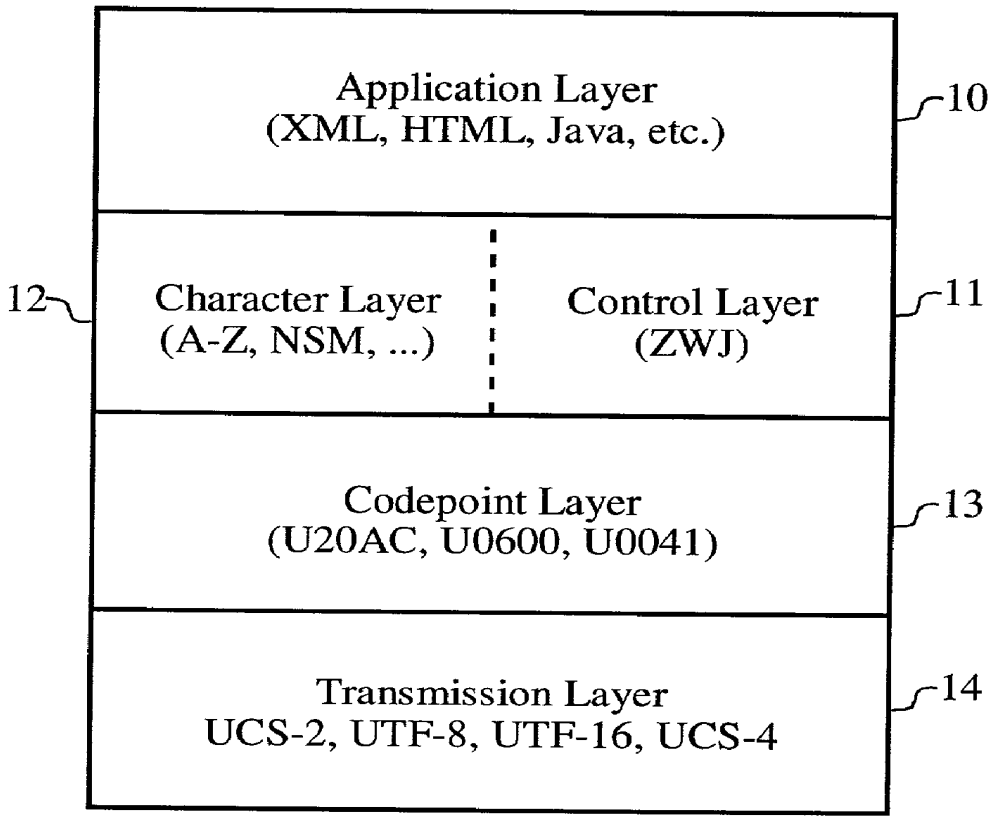


Figure 1

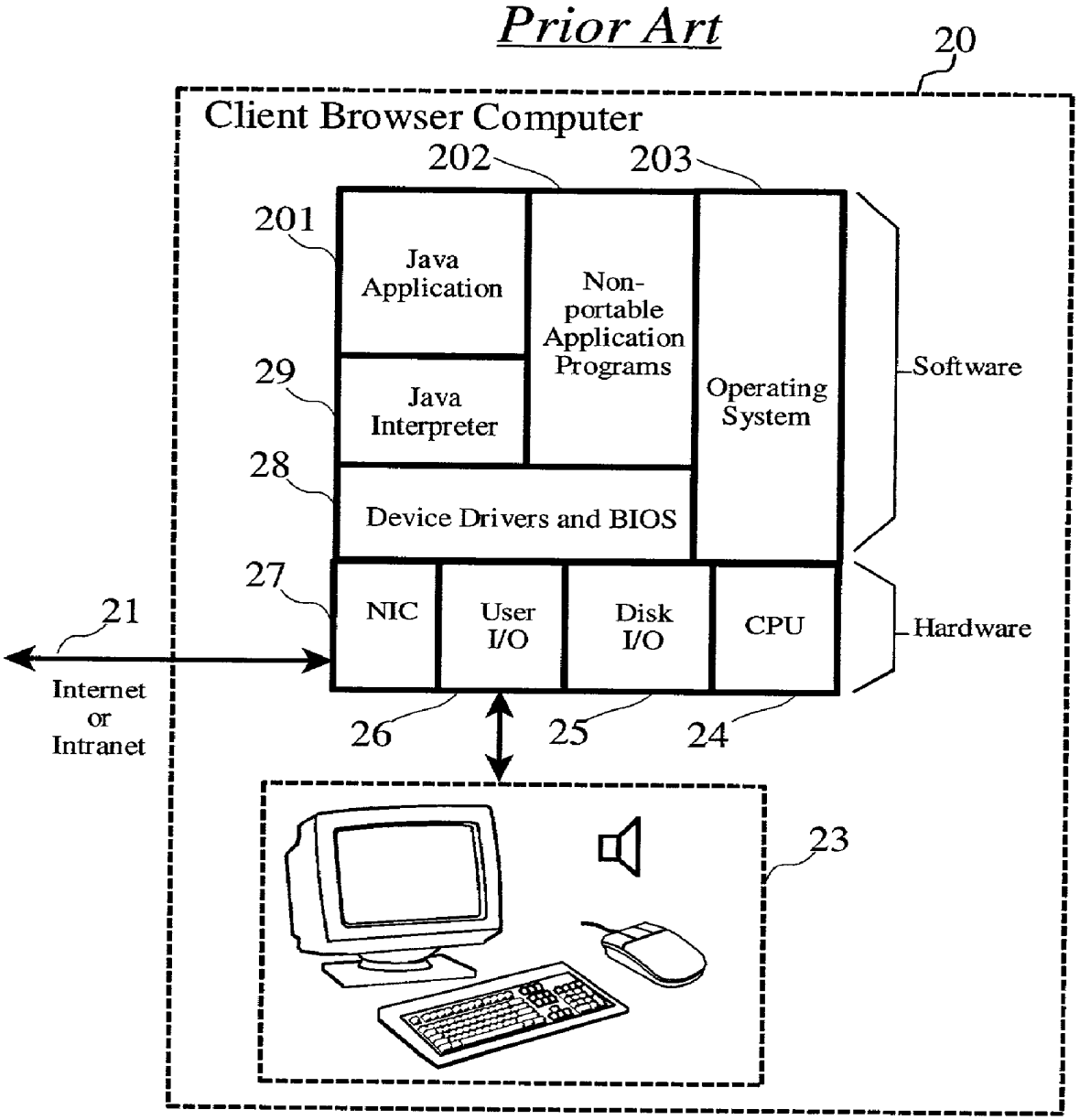

Prior Art
20
Client Browser Computer
202
203
201
Java Application
Non-portable Application Programs
Operating System
Software
29
Java Interpreter
28
Device Drivers and BIOS
27
NIC
User I/O
Disk I/O
CPU
Hardware
21
Internet or Intranet
26
25
24
23

Figure 2

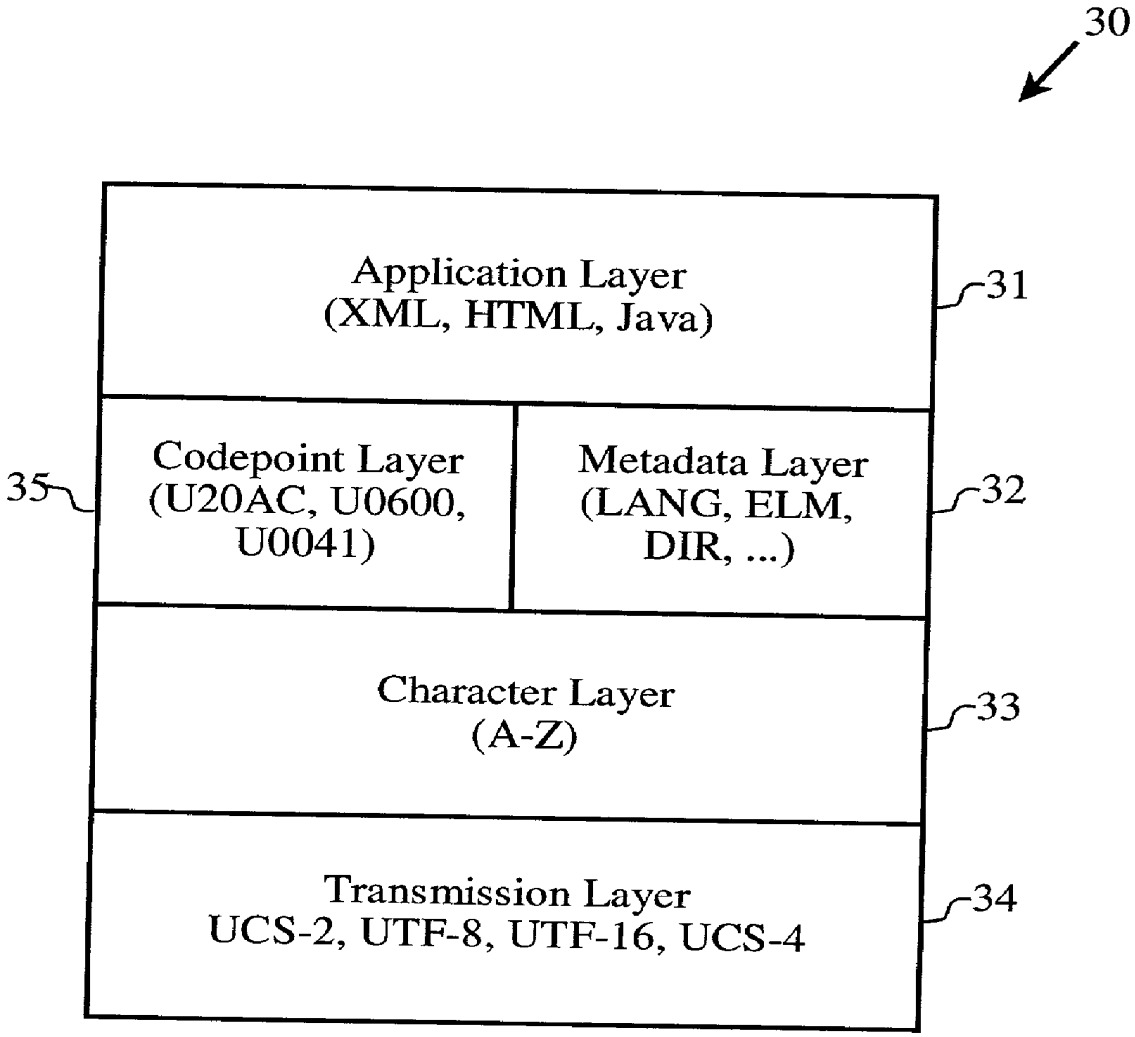

30
Application Layer
(XML, HTML, Java)
31
Codepoint Layer
(U20AC, U0600,
U0041)
35
Metadata Layer
(LANG, ELM,
DIR, ...)
32
Character Layer
(A-Z)
33
Transmission Layer
UCS-2, UTF-8, UTF-16, UCS-4
34

Figure 3

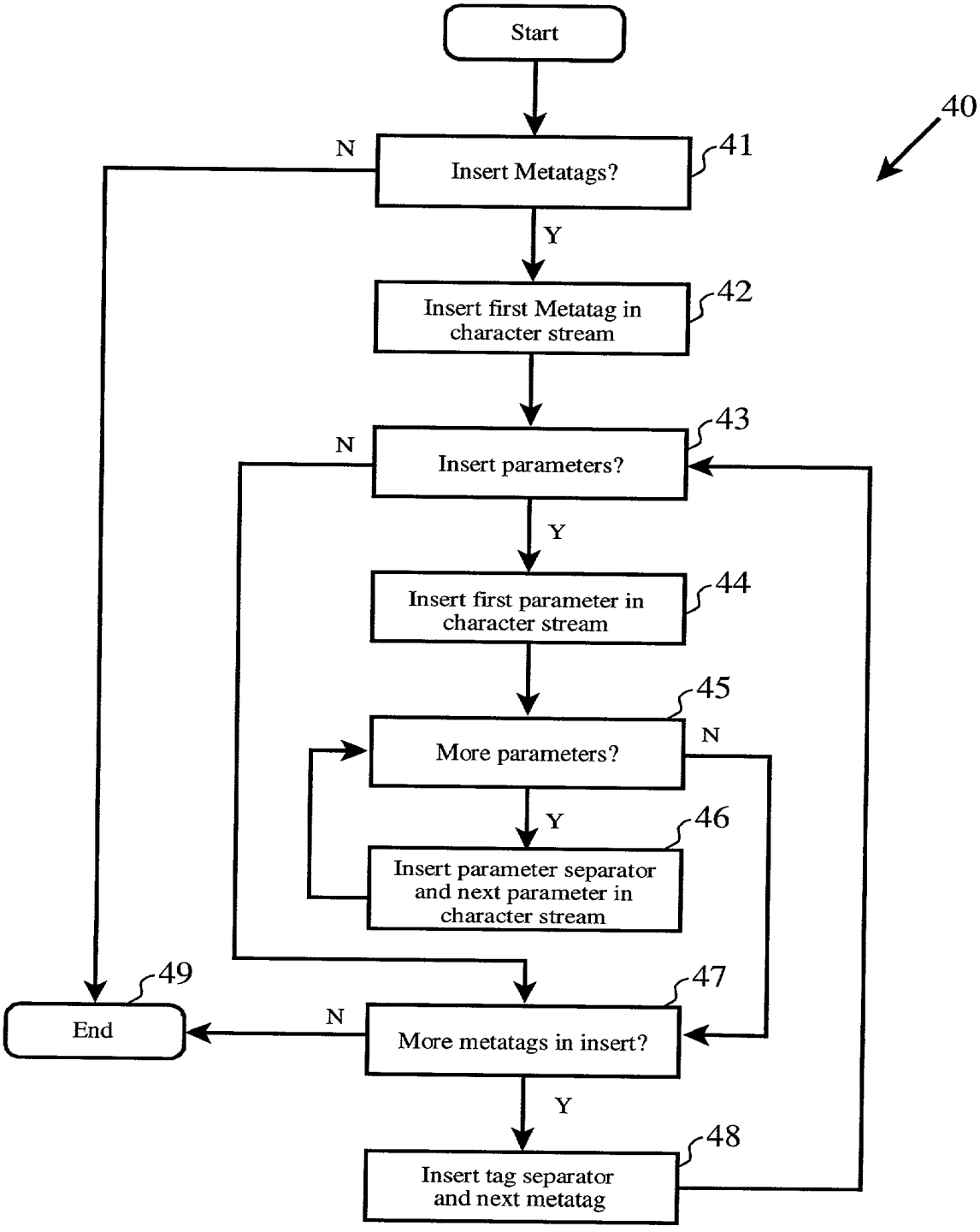

Start
40
N
Insert Metatags?
41
Y
Insert first Metatag in character stream
42
N
Insert parameters?
43
Y
Insert first parameter in character stream
44
More parameters?
45
N
Y
Insert parameter separator and next parameter in character stream
46
End
49
N
More metatags in insert?
47
Y
Insert tag separator and next metatag
48

Figure 4

GENERALIZED MECHANISM FOR UNICODE METADATA

BACKGROUND OF THE INVENTION

[0001] 1. Field of the Invention

[0002] This invention relates to the technologies of computer displays and interpretation of file and data for display on a computer. This invention especially relates to the technologies of universal text encoding, markup languages, and data-to-display methods.

[0003] 2. Description of the Related Art

[0004] The many competing motivations for selecting codepoints within a text encoding standard, such as the Unicode standard, threaten the fundamental purpose of a character encoding: data. Digital data is immensely convenient because the advantages of its great simplicity outweigh the loses incurred by representing knowledge imperfectly.

[0005] Often, in pursuit of all the benefits of such as standard, we set our sights on recovering on what has been left out. For many years, numerical analysts have been systematically improving fidelity of computer models of the apparently continuous world around us. They are helped by the mathematical properties of real numbers. A more difficult challenge is text which represents language.

[0006] In fact, we contend that the ability to interpret raw text has become more difficult. A text stream is no longer just a sequence of agreed upon codepoints. Text manipulation processes require additional information for proper interpretation, such as displaying the encoded text on a computer display or mobile telephone display.

[0007] There has been substantial interest in introducing an architecture for describing language and other semantic information within raw Unicode streams.

[0008] The need for expressing metadata, e.g. information describing data, has existed ever since humans started communicating each other. Prior to written communication, metadata was expressed through our verbal speech. The tone, volume, speed in which something was spoken often signaled its importance or underlying emotion. Often, the metadata may be as significant or even more significant that the data itself, and often much more difficult to codify.

[0009] Writing and printing systems also have a need for metadata. This was conveyed through the use of color, style, size of glyphs. Initially, this metadata was used as a mechanism for circumventing the limitations of early encoding schemes. As our communication mechanisms advanced so did our need for expressing metadata.

[0010] **FIG. 1** presents the Unicode character/control/metadata model, including an application layer **(10)**, a control layer **(11)**, a character layer **(12)**, a codepoint layer **(13)**, and a tranmission layer **(14)**. Unicode is well known in the art, and many alternate representations can be found in widely available literature.

[0011] A primary need for metadata in Unicode occurs in the control layer **(11)**, as one may anticipate. In **FIG. 1, a** dotted line is used to separate the character layer **(12)** from the control layer **(11)** to illustrate the sometimes difficult to define boundary separating characters from control. This inability to provide a clean separation has made the task of developing applications **(10)** that are based on a Unicode more difficult to implement.

[0012] For greater understanding of the present invention, a historical summary is first presented which demonstrates the need for metadata within character encodings. Second, an examination of the presently available paradigms for expressing metadata is provided. In particular, attention is given to both extensible markup language (XML) and Unicode's character/control/metadata model.

[0013] Baudot's 5-bit teleprinter represents one of the earliest uses of metadata Baudot divided his character set into two distinct planes, named Letters and Figures. The Letters plane contained all the Uppercase Latin letters, while the Figures plane contained the Arabic numerals and punctuation characters. These two planes shared a single set of code values.

[0014] To distinguish their meaning, Baudot introduced two special meta-characters, letter shift "LTRS" and figure shift "FIGS". When a sequence of codepoints were transmitted, it was preceded by either the FIGS or LTRS character. This permitted the characters to be interpreted unambiguously. This is similar to the shift lock mechanism in typewriters. For example, line 1 in **FIG. 2** spells out "BAUDOT" while line 2 spells out "?-7$95", as shown in TABLE 1.

TABLE 1

Using LTRS and FIGS in Baudot code	
1	0x1F 0x19 0x03 0x07 0x09 0x18 0x10 BAUDOT
2	0x1B 0x19 0x03 0x07 0x09 0x18 0x10 ?7$95 (2)

[0015] However, this method still left the problem of how to transmit a special signal to a teleprinter operator. Baudot once again set aside a special code point, named bell "BEL". This codepoint would not result in anything being printed, but rather it would be recognized by the physical teleprinter. The teleprinter, having recognized the BEL, character would perform some action, such as ringing of a bell.

[0016] About 1900, metadata characters began to be used as format effectors, such as can be seen in Murray's code. Murray's code introduced two additional characters: (a) column (COL) carriage return in International Telegraphy Alphabet Number 2 (ITA2), and (b) line page (LINE PAGE) line feed in ITA2. These two codes were used to control the positioning of the print wheel, and to control the advancement of paper. This encoding scheme was used for nearly fifty years with little modification. It also served as the foundation for future encoding techniques.

[0017] During the late 1950s and early 1960s, telecommunication hardware rapidly became much more complex. This complexity, however, resulted in the need for more sophisticated protocols, and for greater amounts of metadata. For this purpose, the US Army introduced a 6-bit character code called "FIELDATA." FIELDATA introduced the concept of "supervisor codes", known today has "control codes." These codepoints were used to signal communications hardware.

[0018] The hardware manufacturers were certainly not the only users of metadata, however. It did not take long for the

data processing community to realize that they also had uses for metadata. This unfortunately taxed the existing encoding schemes (5-bit and 6-bit) so much so as to render them unusable, as all of the potential codes to be incorporated to address all of the user needs could not be represented in such a small code space.

[0019] This drove the creation of a richer and more flexible encoding scheme. These issues were directly addressed by the American Standard Code for Information Interchange (ASCII).

[0020] The ASCII code, a 7-bit encoding, served not only as a mechanism for data interchange, but also as an architecture for describing metadata. This metadata could be used for communicating higher order protocols in hardware as well as software. The architecture is based upon ASCII's escape character (ESC) at hex value 0×1B.

[0021] Initially, the ESC was used for shifting to one or more character sets. This was of a particular importance to ALGOL programmers. As ASCII was adopted internationally, the ESC became useful for signaling the swapping in and out of international character sets. This concept was later expanded in 1980s in the International Standards Organization (ISO) ISO-2022 standard.

[0022] ISO-2022 is an architecture and registration scheme for allowing multiple 7-bit or 8-bit encodings to be intermixed. It is a modal encoding system like Baudot. Escape sequences or special characters are used to switch between different character sets or multiple versions of the same character set. This scheme operates in two phases. The first phase handles the switching between character sets, while the second handles the actual characters that make up the text.

[0023] Non-modal encoding systems make direct use of the byte values in determining the size of a character. In such a scheme, characters may vary in size within a stream of text, typically ranging from one to three bytes. This can be witnessed in the well-known UTF-8 and UTF-16 encodings.

[0024] In ISO-2022, up to four different sets of graphical characters may be simultaneously available, labeled G0 through G3. Escape sequences are used to assign and switch between the individual graphical sets. For example, line 1 in TABLE 2 shows the byte sequence for assigning the ASCII encoding to the G0 alternate graphic character set. Line 2 of TABLE 2 shows the Latin-1 encoding being assigned to the G1 set.

TABLE 2

Example ISO-2022 Escape Sequences		
1	ESC 0x28 0x42	assign ASCII to G0
2	ESC 0x2D 0x41	assign Latin 1 to G1

[0025] Most data processing tools make little if any distinction amongst data types. The only distinctions being purely human user interpretation. Data is simply viewed by the processing tools in terms of bytes. For example, the common UNIX text searching utility known as GREP assumes that data is represented as a linear sequence of stateless fixed length independent bytes. GREP is highly flexible when it comes to searching, whether it be characters or object code. This model has served well under the assumption that one character equals one codepoint, but encoding systems have advanced and user expectations have risen.

[0026] Over the last ten or so years, Unicode has become the defacto standard for encoding multilingual text. This has brought a host of new possibilities that only few could have previously imagined. Users however, want more than just enough information for intelligible communication. Plain text in its least common denominator is simply insufficient.

[0027] There have been several discussions concerning the enrichment of plain text of which ISO-2022 is one. Even XML can be viewed in this framework. Both concern meta information yet have different purposes, goals, and audiences. The transition from storing and transmitting text as plain streams of code-points is now well underway.

[0028] Extensible markup language (XML) provides a standard way of sharing structured documents, and for defining other markup languages. XML uses Unicode as its character encoding for data and markup. Control codes, data characters, and markup characters may appear intermixed in a text stream.

[0029] When this situation is combined with overlapping mechanisms for encoding higher order information, confusion and ambiguity may ensue when processing or interpreting the encoded data There may exist situations in which markup and control codes should not be interleaved. This issue is quickly coming to realization within XML and Unicode.

[0030] Whitespace characters in XML are used in both markup and data. The characters used in XML to represent whitespace are limited to "space", "tab", "carriage return", and "line feed". Unicode, on the other hand, offers several characters for representing whitespace. In particular, the line separator U2028 and the paragraph separator U2029. Their use however within XML may lead to ambiguities due to the additional implied semantics.

[0031] In Unicode, these characters may be used to indicate hard line breaks and paragraphs within a stream. These may affect visual rendering, as well as serve as separators. When used within XML, however, it is unclear whether the implied semantics can be ignored. Does the presence of one of these control codes indicate that a rendering protocol is being specified in addition to their use as whitespace, or are they simply whitespace?

[0032] The use of name "tags" within XML also posses problems. The characters in the Compatibility Area and Specials Area UF900-UFFFE from Unicode are not permitted to be used in names within XML.

[0033] Their exclusion is due in part to the characters being already encoded in other places within Unicode. By no means, though, is this the only reason. If characters from the Compatibility Area were included, the issue of normalization would then need to be addressed. In this context normalization refers to names being equivalent, but not necessarily the same. Additionally, characters that pose both a decomposed and precomposed form also need attention.

[0034] Unicode attempts to address these issues in Unicode Technical Report #15 "Unicode Normalization Forms", which is freely available from the Unicode orga-

nization. Unicode provides guidelines and an algorithm for determining when two character sequences are equivalent. In general, there are two classes of normalization: Canonical and Compatibility.

[0035] Canonical normalization handles equivalence between decomposed and precomposed characters. This type of normalization is reversible. Compatibility normalization addresses equivalence between characters that visually appear the same, and is irreversible.

[0036] Compatibility normalization in particular is problematic within XML. XML is designed to represent raw data free from any particular preferred presentation. Characters that may be compatible for presentation purposes, however, do not necessarily share the same semantics. It may be the case that an additional protocol is being specified within the stream. For example, the UFB0 character on line 1 TABLE 3 is compatible with the two character sequence "U0066 U0066" on line 2. Line 1 however, also specifies an additional protocol: ligatures. In such a situation, it is unclear whether or not the names were intended to be distinct. It is difficult to tell when the control function (higher order protocol specification) of a character can be ignored and when it can not.

TABLE 3

Example Compatibility Normalization Ambiquity

1	UFB00	ff ligature
2	U0066 U0066	ff no ligature

[0037] Further, some have argued that Unicode's Normalization Algorithm is difficult to implement, resource intensive, and prone to errors. To avoid such problems XML has chosen not to perform normalization when comparing names.

[0038] Problems such as these are due to the lack of separation of syntax from semantics within Unicode. The absence of a general mechanism for specifying protocols "metadata" only serves to confound these issues even further.

[0039] There are two well-known general approaches to encoding metadata within text streams: in-band signaling and out-of-band signalling. Inband signalling conveys metadata and textual content using a single shared set of characters, while out-of-band signalling conveys metadata independently from the data. In-band signalling is employed within hyper text markup language (HTML) and XML.

[0040] Determining whether a character is data or metadata using in-band-signalling depends on the context in which a character is found. That is, code points are "overloaded." This achieves maximal use of the character encoding, as characters are not duplicated. It also does not require encoding modifications as protocols change.

[0041] All of this, however, comes at the expense of the complexity of parsing the data. It is no longer possible to conduct a simple parse of a stream looking for just data or metadata.

[0042] Using out-of-band signalling for describing Unicode metadata requires the definition and transmission of complex structures serving a similar purpose as document data type definitions (DTD) in XML. This has the ill effect of making the transmission of Unicode more intricate. It would no longer be acceptable to simply transmit the raw Unicode text. Without the metadata, the meaning of the raw text may be ambiguous. On the other hand, parsing of data and metadata may be trivial, given that the two are not intermixed. The transmission problems requiring pairs of raw data files and metadata files to be handled together often may outweigh the potential parsing benefits of out-of-band signalling, depending on the application.

[0043] It is still possible to construct a metadata signalling mechanism for the specific purpose of mixing data and metadata and yet allows for simple parsing. This is the approach that is currently under discussion within the Unicode community and can be found in Unicode Technical Report #7. It is called "light-weight in-band signalling".

[0044] According to this proposed approach, this is achieved in Unicode through the introduction of a special set of characters that may only be used for describing metadata "tagging". The current model under consideration within Unicode is to add 97 new characters to Unicode. These characters would be comprised of a copy of the ASCII graphic characters, a language character tag, and a cancel tag character. These characters would be encoded in Plane 14 "surrogates" U000E0000—U000E007F. These characters could then be used to spell out any ASCII based metadata protocol which needs to be embedded within a raw Unicode stream of text. This permits the construction of simple parsers for separating metadata from data since there is no overloading of characters.

[0045] The use of the tags is very simple. First, a tag identifier character is chosen, followed by an arbitrary number of unicode tag characters. A tag is implicitly terminated when either a non tag character is found or another tag identifier is encountered. Currently there is only one tag identifier defined, the "language" tag, as shown in TABLE 4. Line 1 in TABLE 4 demonstrates the use of the fixed codepoint language tag "U000E0001", along with the cancel tag "U000E007F". The plane 14 ASCII graphic characters are in bold and are used to identify the language. The language name is formed by concatenating the language ID from ISO-639 and the country code from ISO-3166. In the future, a generic tag identifier may be added for private tag definitions.

TABLE 4

Example Unicode Light-Weight
In-band Signaling Language Tag

U000E0001 fr-Fr french text U000E0001 U000E007F

[0046] Tag values can be cancelled by using the tag cancel character. The cancel character is simply appended onto a tag identifier. This has the effect of cancelling that tag identifier's value. If the cancel tag is transmitted without a tag identifier the effect is to cancel any and all processed tag values.

[0047] The value of a tag continues until either it implicitly goes out of scope or a cancel tag character is found. Tags of the same type may not be nested. The occurrence of two consecutive tag types simply applies the new value to the

rest of the unprocessed stream. Tags of differing types may be interlocked. Tags of different types are assumed to ignore each other. That is there are no dependencies between tags.

[0048] Tag characters have no particular visible rendering and have no direct affect on the layout of a stream. Tag aware processes may chose to format streams according to their own interpretation of tags and their associated values. Tag unaware processes should leave tag data alone and continue processing.

[0049] Although, the general light-weight approach to metadata definition is useful, it however posses two problems. First, new tag identifiers always require the introduction of a new Unicode codepoint. This puts Unicode as a standard in a constant state of flux, as well as fixing or limiting the number of possible tag identifiers. Second, there is no method to specify multiple parameters for a tag. This deficiency forces the creation of additional tag identifiers to circumvent this limitation.

[0050] As these specific illustrations and cases indicate, the handling of character data in information processing has always been troublesome. Small encoding mechanisms limit the potential trouble. Many compromises take place completely outside the character set while encoding the data.

[0051] On the other hand Unicode has enough space for lots of problems. This trouble has largely been centered around the inability to clearly separate the notions of syntax, semantics, and protocols.

[0052] The many demands placed on codepoints from Unicode has led to confusion in areas of text exchange, legacy interchange, glyph picking, and others. This confusion has intimidated adopters into non-conformance, consider Unicode normalization within XML and Java.

[0053] Therefore, there is a need in the art for a method and system which allows the present collection of convoluted, unused, and unimplementable Unicode algorithms to be recast in a more manageable context, and which allows the algorithms to become detectable, reversible as well as convertible. Further, there is a need in the art for this new method and system to provide extensibility to Unicode, such as is available in markup languages such as XML, without requiring new tag identifiers to be registered by a protocol controlling authority. Additionally, there is a need in the art for this new method and system to allow for an arbitrary number of control parameters to be specified in a data stream.

BRIEF DESCRIPTION OF THE DRAWINGS

[0054] The following detailed description when taken in conjunction with the figures presented herein provide a complete disclosure of the invention.

[0055] **FIG. 1** shows the layered organization of Unicode.

[0056] **FIG. 2** shows the well-known organization of hardware computing platforms capable of executing Java programs or applets.

[0057] **FIG. 3** shows the layered organization of the invention.

[0058] **FIG. 4** illustrates the logical flow of encoding metatags and parameters into character streams according to the invention.

SUMMARY OF THE INVENTION

[0059] A general mechanism and process for including metadata within the Unicode framework is provided by the invention which is both flexible and extendable. The present invention allows Unicode to simply provide a mechanism for specifying higher order protocols, instead of embedding control functionality under the guise of characters. According to the new model, metadata is always distinct from character data. A provided tag mechanism allows for an unlimited number of possible identifiers, yet does not require any future codepoints to be registered by a standardization body or entity.

[0060] By adopting the framework of the invention, Unicode is freed to deal entirely with the definition of characters, which affords the greatest level of flexibility while still retaining the ability to perform simple parsing.

[0061] The present collection of convoluted, unused, and unimplementable algorithms (Bidi, Normalization, Line Breaking, etc.) can be recast in a more manageable context according to the metadata framework of the invention. The algorithms become detectable, reversible as well as convertible, as a result.

[0062] Further, through use of the invention, there is no longer any need for the traditional Unicode Control Layer. The syntax of controls are captured universally by the Metadata Layer, irrespective of whether the control relates to presentation or content. The indistinct boundary separating characters and control is now replaced by a well defined clear line. This precise separation makes applications that rely on Unicode easier to write and maintain.

[0063] As such, the invention provides a fully open extendable metadata mechanism in which complex semantics can be expressed through simple metadata tags.

DETAILED DESCRIPTION OF THE INVENTION

[0064] The invention provides an enhanced method for handling metadata associated with encoded text data through a number of changes and improvements to the Unicode "light-weight in-band signalling" (LWIB) method. It is preferably implemented in Java, but may equally well be implemented in any other suitable language.

[0065] The invention is realized in part by a computing platform, such as an IBM-compatible personal computer, Apple MacIntosh [TM], or other computer hardware platform, running a common operating system such as Linux, UNIX, Microsoft's Windows [TM], IBM'SAIX[TM] or OS/2 [TM]. According to the preferred embodiment, the method is encoded in the Java programming, which can be executed by many computing platforms suitably equipped with one of several widely-available Java interpreters, or compiled from Java to machine-specific executable code.

[0066] Turning to **FIG. 2, a** generalized organization of such a computer platform **(20)** is shown. The computer platform **(20)** has a central processing unit (CPU) **(24)**, a set of device drivers and a basic input/output system (BIOS) **(28)**, and typically an operating system **(203)**, such as mentioned previously. Most computer platforms, such as a personal computer, are also equipped with disk interfaces **(25)** and disks; user device I/O **(26)** to interface to key-

boards, pointing devices, and a display; and a network interface card or device **(27)** allowing communications to a computer network, wireless network, or the Internet. Some computer platforms, such as personal digital assistants, web-enabled telephones, and Internet appliances may not be provided with all of these components, but in general, the functionality of these components is present in some form.

[0067] The computer platform **(20)** is also typically provided with one or more non-portable, machine-specific application programs **(202)**.

[0068] According to the preferred embodiment, the computer platform is provided with a Java interpreter **(201)**, which are freely available for a variety of operating systems and computer platform, and which are well-known in the art.

[0069] The remaining disclosure of the invention is presented relative to the computer program implementation of the method for incorporating and interpreting metadata embedded into Unicode data streams.

[0070] Turning to **FIG. 3**, the new layered organization **(30)** of the invention is shown, including an application layer **(31)**, character layer **(35)**, metadata layer **(32)**, codepoint layer **(33)**, and transmission layer **(34)**. Unlike the general Unicode model, this new model provides for a distinctly separate character layer **(35)** and metadata layer **(32)** through the use of metadata embedded in Unicode data, resolving the otherwise ambiguous definition of character and control codes.

[0071] First, the method keeps the copy of the ASCII graphic characters and the cancel tag, as in the Unicode LWIB, but it omits the fixed codepoint tag identifiers. In their place, two new characters are employed, a tag separator U000E0001 and a tag argument separator U000E0002, as shown in TABLE 5.

TABLE 5

New Characters in the Method of the Invention

Tag Characters	UCS-4	Visual Representation
separator	U000E0001	|
argument separator	U000E0002	=
cancel	U000E007F	~
space	U000E0020	
graphic characters	U000E0021-U000E007E	a–z, A–Z , 0–9, etc.

[0072] Use of these new characters is similar to the methods employed by SGML, XML, and HTML. As such, these new characters provide an easy migration path for embedding XML-like protocols within Unicode. The use of these characters is by no means required—higher applications may chose alternative methods.

[0073] The tag separator character is used to separate consecutive tags from one another, while the tag argument separator is used to delineate multiple arguments of a tag. This aspect of the invention allows the same characters to be used for tag values as well as tag identifiers. Further, tag identifiers are spelled out, rather than being assigned to a fixed single codepoint.

[0074] After all the parameters for the first metatag are insterted in the character stream, subsequent metatags are inserted separated by tag separators **(47, 48)**, each being followed any parameters and parameter separators as needed.

[0075] This allows the use of tags to remain simple. First, the tag is spelled out using the ASCII tag characters, followed by a tag argument separator. This provides for an arbitrary number of tag values for a tag identifier, each being separated by a tag argument separator.

[0076] A tag identifier is terminated by either encountering a tag argument separator, a tag separator, or a non-tag character. This still allows for relatively simple parsing.

[0077] Turning to **FIG. 4**, the fundamental logic flow of the invention for encoding metatags into Unicode data is shown. If metatags are to be inserted into the Unicode data **(41)**, then the first metatag is inserted **(42)** in the character stream accordingly. If any parameters are to be included with the first metatag **(43)**, then the first parameter is inserted **(44)** following the metatag. If more than one parameter is to be inserted following the first metatag **(45)**, these parameters are inserted following their metatag, separated by parameter separators **(45, 46)**. Decoding of this encoding scheme is done by following the reverse process, first finding (instead of inserting) a first tag, then parsing for parameters followed by parameter separators, and the parsing for subsequent metatags and parameters.

[0078] In the following disclosure, tag characters are represented enclosed in braces "{}", the vertical bar character "|" depicts the tag separator, the equals sign "=" denotes the tag argument separator, and the tilde "~" will represent the tag cancel character. For example, line 1 in TABLE 6 **FIG. 6** shows a stream with two embedded tags "XX" and "YY". In this example, the tag "XX" has one argument "a", while the "YY" tag has two arguments "b" and "c".

TABLE 6

An Example of the Data using the New Metatag Method

def{XX}={a}|{YY}={b}={c}ghi~{YY}jkl~{XX}

[0079] The example suggests the nesting of "YY" within "XX". The semantics of such combinations are left to protocol designers rather than the metadata. This affords the greatest flexibility, and yet still retains the ability to perform simple parsing.

[0080] Further, this method allows a standardization body such as Unicode to simply be in the business of defining mechanism rather than mechanism and policy. It is possible that a standards body such as Unicode could act as the "registrar" of new tag identifiers while working in conjunction with other standards bodies. This however, does not preclude private tags from being defined for those cases in which widespread protocol adoption is not required, which is another advantage of the present invention.

[0081] According to another advantage and characteristic of the invention, the semantics of the cancel tag will may be left to a protocol designer. It is possible that in some protocols the cancel tag might "undo" the last tag, while in others, it may act as a end marker for terminating scope. Additionally, there is no requirement that a cancel tag be used at all.

[0082] The example of TABLE 7 shows how the language tag may be represented in the new tagging model of the invention. Line 1 in TABLE 7 is copied from TABLE 4 for reference, and line 2 of TABLE 7 shows the language tag spelled out with the two tag arguments being clearly delineated. The spelling out of tag identifiers is a negligible processing requirement when balanced against the flexibility and advantages of this method.

TABLE 7

	Example of "Spelling Out" the Language Tag
1	000E0001 fr-FR french text U000E0001 U000E007F
2	{LANG}={fr}={FR} french text~{LANG}

[0083] Currently, the Unicode reference Bidirectional Algorithm treats tag characters as having the property of left-to-right. At first, this does not seem problematic as the tags and the tag values should always be interpreted as left-to-right. Unfortunately the tags may inadvertently influence the resolution of weak and neutral types due to their juxtaposition. The example in TABLE 8 demonstrates this error.

[0084] In TABLE 8, Arabic characters are represented in upper case. Line 1 is a sequence of characters in logical order, line 2 is the expected resultant display ordering, and line 3 is the actual rendered display ordering.

TABLE 8

	Example Error in Bidirectional Processing
1	CIBARA {LANG}={ar}={EG}, 123
2	123, {LANG}={ar}={EG} ARABIC
3	{LANG}={ar}={EG}, 123 ARABIC

[0085] The display ordering on line 3 of TABLE 8 is incorrect because the tag characters inadvertently participated in bidirectional processing. This problem is solved by introducing another new bidirectional property, "ignore", according to the present invention.

[0086] This enables the Bidirectional Algorithm to continue to function properly, while also protecting the semantics of tags. Characters that possess the "ignore" type do not have any direction. These characters are prevented from participating in the Unicode Bidirectional Algorithm.

[0087] Traditionally, text processes manipulated ASCII data with the implicit understanding that every codepoint equated to a single character and in turn a single text element, which then served as a fundamental unit of manipulation. In most cases this assumption held, especially given that only English text was being processed.

[0088] Multilingual information processing, however breaks the assumption that codepoints, characters, and text elements are all equal. Text elements are directly tied to a text process, script, and language. Common encodings today provide an abstract set of characters directly mapped onto set of numerals. The abstract characters are then grouped to form text elements.

[0089] In some cases, a text element may still equate to a single character, while in other situations, a text element may be comprised of several characters. For example, in Spanish the character sequence "ll" is treated as a single text element when sorted, but is treated as two text elements "l" and " when printed.

[0090] Unicode relies on an abstract notion of characters and text elements. Unfortunately, a general mechanism for indicating text elements is lacking. In some instances a text element is implicitly specified through a sequence of characters. For example, line 1 in TABLE 9 shows how a base character and a non spacing diacritic combine to form a single text element, line 2.

TABLE 9

	Example Unicode Character Combining
1	U00D6 Ö decomposed
2	U004F U0308 Ö precomposed

[0091] In other cases, text elements are explicitly specified by control codes. In particular, Unicode uses control codes for forming visual text elements: the zero width joiner U200D and the zero width non joiner U200C control codes. These characters affect ligature formation and cursive connection of glyphs. The intended semantic of the zero width non joiner is to break cursive connections and ligatures. The zero width joiner is designed to form a more highly connected rendering of adjacent characters.

[0092] For example, line 1 in TABLE 10 shows the sequence of codepoints for Unicode constructing a ligature. The characters x and y represent arbitrary characters. Line 2 shows how the zero width non joiner can be used to break a cursive connection. However, problems arise when one wishes to suppress ligatures while still promoting cursive connections. In this situation, Unicode recommends combining the zero width nonjoiner and the zero width joiner, such as shown in line 3 **FIG. 10**.

TABLE 10

	Example Unicode Joiners
1	x U200D y
2	x U200C y
3	x U200D U200C U200D y

[0093] Rather than using control codes with complicated semantics and implicit sequences of characters to form text elements, a simple generalized mechanism is provided by the present invention. Because Unicode has no general way to indicate that sequences of characters should be viewed as a single text element, the currently approach in the art relies on a a higher order protocol outside of Unicode, such as XML. The trouble in taking such approach is that it is ill suited for this purpose. XML is designed to describe the structure of documents and collections of data not individual characters and text elements. XML requires data to strictly adhere to a hierarchical organization. This may be appropriate for documents, but may be troublesome for a simple text stream.

[0094] The model that is really required needs to be organized around characters and text elements, as is provided by the present invention. This is achieved through

metadata tags and simple protocols. For example, the zero width joiner and zero width non joiner characters can be described by a new tag, such as text element "ELM", using the new method. Then, the ELM tag can be used to group multiple characters together so that they can be treated as a single grapheme or text element. For example, line 1 in TABLE 11 shows a text element "xy" for all purposes.

TABLE 11

Example use of Invention to Define a Text Element Tag	
1	{ELM}xy~{ELM}
2	{ELM}={LIG}xy~{ELM}
3	{ELM}={JOIN}xy~{ELM}
4	{ELM}={COLL}ch~{ELM}
5	{ELM}={CASE}SS~{ELM}

[0095] When characters are grouped together it may be for the purpose of rendering, sorting, or case conversion. The purpose of the grouping does not need to be understood by Unicode. The semantics should only be determined by processes that make direct use of such information. The tag is simply a mechanism for signaling higher order semantics.

[0096] For example, line 2 in TABLE 11 shows a text element "xy" for the purposes of forming ligatures, but not searching/sorting, and line 3 demonstrates the text element "xy" being cursively connected while yet suppressing ligature formation.

[0097] Additionally the new ELM tag can be used to form other semantic groupings. For example, in Spanish when "c" is followed by "h", the two single characters combine to form the single text element "ch", such as shown in line 4 of TABLE 11. This grouping does not effect rendering, but has implications in sorting. In German however, groupings affect case conversion. For example, the character sequence "SS" when converted to lowercase results in the single etset character "β", such as in line 5 of TABLE 11.

[0098] As such, plain text streams that contain characters of varying direction pose a particular problem for determining the correct visual presentation. There are several instances in which it is nearly impossible to render bidirectional text correctly in the absence of any higher order information. In particular, picking glyphs requires that a rendering engine have knowledge of fonts.

[0099] The Unicode Bidirectional Algorithm operates as a stream to stream conversion. At first, this seems fine given that Unicode is a character encoding mechanism and not a glyph encoding scheme. This output, however is insufficient by itself to correctly display bidirectional text. If a process is going to present bidirectional text, then the output needs to be glyphs and glyph positions. This presents a problem for Unicode. The Unicode Bidirectional algorithm can not possibly produce this output and yet still remain consistent with Unicode's overall design goals, that of a character encoding scheme.

[0100] Unicode's algorithms should only be based on character attributes and codepoints. By introducing metadata according to the invention, however, the improved Unicode would permit a cleaner division of responsibilities. Algorithms could be recast to take advantage of this division. In particular, the output of the Bidirectional Algorithm could be changed to raw Unicode with embedded metadata "tags". This would separate the responsibility of determining directional boundaries from glyph picking.

[0101] The core of the reference Unicode Bidirectional algorithm is centered around three aspects: resolving character types, reordering characters and analyzing mirrors. The bidirectional algorithm is applied to each paragraph on a line by line basis. During resolution, characters that do not have a strong direction are assigned a direction based on the surrounding characters or directional overrides. In the reordering phase, sequences of characters are reversed as necessary to obtain the correct visual ordering. Finally each mirrored character (parenthesis, brackets, braces, etc.) is examined to see if it needs to be replaced with its symmetric mirror.

[0102] Unfortunately, this method has the effect of making an irreversible change to the input stream. The logical ordering is no longer available. This inhibits the construction of an algorithm that takes as input a stream in display order and produces as output its corresponding logical ordering. The example in TABLE 12 illustrates this problem. In TABLE 12, Arabic letters are depicted by upper case latin letters while the right square bracket "[" indicates a right to left override U202E. In TABLE 12, line 1 is a stream in display order, and lines 2 and 3 are streams in logical order. If the bidirectional algorithm is applied to line 2 or line 3, the result is line 1 in either case.

TABLE 12

Example Mapping from Display Order to Logical Order	
1	123 (DCBA)
2	(ABCD) 123
3	]123 (ABCD)

[0103] It is also impossible to tell whether a stream has been processed by the Bidirectional Algorithm. The output does not contain any identifying markers to indicate that a stream has been processed. This makes the transmission of bidirectional data problematic. A process can never be sure whether an input stream has undergone bidirectional processing. To further complicate the situation the bidirectional algorithm must be applied on a line by line basis. This is not always easy to accomplish if display and font metrics are not available.

[0104] In this paper we propose the introduction of three tags for bidirectional processing: "PAR" paragraph, direction "DIR", and mirror "MIR".

[0105] The PAR tag signifies the beginning of a paragraph. It takes one argument, the base direction of the paragraph either right "R" or left "L".

[0106] The DIR tag takes one argument as well, the resolved segment's direction either "L" or "R".

[0107] The MIR tag does not require any argument. Its presence indicates that the preceding character should be replaced by its symmetric mirror. The scope of the DIR tag is terminated by either a cancel tag, a PAR tag, or the end of the input stream.

[0108] For example, in TABLE 13, line 1 represents a stream of characters in logical order and Line 2 is the output stream after running the bidirectional algorithm using tagging. Arabic letters are represented by upper case latin letters, and tag characters are enclosed in brackets "{}". Again, the equal sign represents the tag argument separator, the vertical bar represents the tag separator "U000E0001", and tilde represents the cancel tag character. The output of the algorithm only inserts tags to indicate resolved directional boundaries and mirrors. The data characters still remain in logical order.

TABLE 13

Example Bidirectional Processing Using Metatags of the Invention

1	(ABCD) 123
2	{PAR}={R}|{MIR}(ABCD{MIR}) {DIR={L}|123~{DIR}|~{PAR}|

[0109] Furthermore, the bidirectional standard Unicode embedding controls "LRE","RLE", "LRO", "RLO", and "PDF" can be eliminated because they are superseded by the DIR tag. These controls act solely as format effectors. They convey no other semantic information and are unnecessary when viewed in light of the DIR tag.

[0110] The introduction of these new tags does not require a re-implementation of the entire standard Unicode Bidirectional Algorithm, however. The method only requires two changes to accommodate the new tags. In those places where the text is to be reversed, a DIR tag is inserted to indicate the resultant direction rather than actually reversing the stream itself. In those places where a symmetric mirror is required, a MIR tag is inserted to indicate that this character should be replaced with its corresponding mirror.

[0111] According to the preferred embodiment, the invention's Javafunctions "taglevel" and "tagrun" shown in TABLE 17, lines 1 through 45, replace functions "reverseRun", "reverseLevels" and "reorder" in the reference Unicode method. The mirror function has been changed to insert a MIR tag rather than directly replacing a character with its symmetric mirror.

[0112] The Bidirectional Algorithm may also be extended to directly interpret tags itself. This would be extremely beneficial in cases where the data and the implicit rules do not provide adequate results. For example, in Farsi, mathematical expressions are written left to right while in Arabic they are written right to left. [0111] Under the standard reference Bidirectional Algorithm, control codes would need to be inserted into the stream to force correct rendering, such as shown in line 1 of TABLE 14 where the characters "LRE" and "PDF" represent the Unicode control codes Left to Right Embedding and Pop Directional Format respectively.

TABLE 14

Example Mathematical Expression

1	LRE 1 + 1 = 2 PDF
2	{LANG}={fa}={IR}|{MATH} 1 + 1 = 2~{MATH}
3	{LANG}={fa}={IR}|{MATH}|{DIR}={L} 1 + 1 = 2 ~{MATH}|~{DIR}

[0113] The extended Bidirectional Algorithm of the invention may address this through the addition of two tags: "MATH" and "LANG". These tags may be inserted into the stream to identify the language and that portion of the stream that is a mathematical expression. By using the tagging method of the invention, the output stream still remains in logical order with its direction correctly resolved without the need of control codes, such as shown in lines 2 and 3 of TABLE 14.

[0114] Turning to HTML for application of the invention, the HTML 4.0 specification introduces a bidirectional override tag "BDO" for explicitly controlling the direction by which a tag's contents should be displayed. Lines 1 and 2 in TABLE 15 illustrate the syntax of this tag.

TABLE 15

Example of HTML BDO Tag Usage

1	<bdo dir="LTR">body content</bdo>
2	<bdo dir="RTL">body content</bdo>

[0115] These HTML tags can be used in conjunction with the Unicode bidirectional tags through the method of the present invention. The Unicode tags can be directly converted into the HTML bidirectional tags. This allows for a clean division of responsibilities for displaying bidirectional data.

[0116] The Unicode metadata tags simply serve as bidirectional markers. Browsers can then directly render the resultant HTML. This permits the Unicode bidirectional algorithm to be free from the problems of determining font and display metrics.

[0117] The UniMeta program, presented in TABLE 18, takes as input a file encoded in UTF-8 which contains Unicode text in logical order with bidirectional tags, in lines 1-105. The UniMeta program then converts the input text into HTML. Each Unicode metadata tag is replaced with a corresponding HTML tag.

[0118] Currently, there is no corresponding tag for mirroring in HTML. When a Unicode MIR tag is found, it is simply ignored. The example in TABLE 16 illustrates the output from the UniMeta Java program. Lines 1 and 2 are copied from TABLE 13, and line 3 is the resultant HTML with BDO tags.

TABLE 16

Example Input and Output from UniMeta

```
1  (ABCD) 123
2  {PAR}={R}|{MIR}(ABCD{MIR}) {DIR}={L}|123~{DIR}|~{PAR}|~
3  <bdo dir="rtl">(ABCD)<bdo dir="ltr">123</bdo></bdo>
```

[0119] By using metadata tags with a Bidirectional Algorithm, a clear division of responsibilities is achieved. The bidirectional layout process is now divided into two separate and distinct phases, logical run determination and physical presentation. This permits character data to remain in logical order, yet still contain the necessary information for it to be correctly displayed. Additionally, any text process receiving such a stream is able to immediately detect that the stream has been bidirectionally processed.

[0120] As it will be recognized by those skilled in the art, the metadata model and method of the invention described herein is adaptable to other systems and other types of information, and thus is not limited to the specific examples disclosed herein. For example, when used in source programming languages, metadata characters could be used to indicate comments, enabling metatag-aware compilers to know exactly where comments were independent of context of the source code, making it unnecessary to have special comment starters, enders, or rules concerning their usage. This would also enable the development of tools such as "javadoc" in a more straightforward streamlined fashion. In fact, such tools could be written independent of language given that comments would always be expressed via metadata.

[0121] Furthermore, text processing languages such as TeX could be improved to incorporate the metadata methods disclosed herein. This would enable the creation of such tools as universal spell checkers. No longer would a spell checker need to understand the syntax of TeX commands. All TeX commands would simply be expressed through metadata.

[0122] As such, it will be recognized by those skilled in the art that many variations, alternate embodiments, and applications of the invention may be made without departing from the spirit and scope of the invention. Thus, the scope of the invention should be determined by the following claims.

TABLE 17

Example Java Source Code for BiDi Implementation

```
1 --Unicode metadata tags
2 dirL=map intToWord32 [0xe0044,0xe0049,0xe0052,
3 0xe0002,0xe004c,0xe0001]
4 dirR=map intToWord32 [0xe0044,0xe004c,0xe0052
5 0xe0002,0xe0052,0xe0001]
6 dirEnd=map intToWord32 [0xe007f,0xe0044,0xe0049
7 0xe0052,0xe0001]
8 parL=map intToWord32 [0xe0050,0xe0041,0xe0052,
9 0xe0002,0xe004c,0xe0001]
10 parR=map intToWord32 [0xe0050,0xe0041,0xe0052,
11 0xe0002,0xe0052,0xe0001]
12 parEnd=map intToWord32 [0xe007f,0xe0050,0xe0041
13 0xe0052,0xe0001]
14
```

TABLE 17-continued

Example Java Source Code for BiDi Implementation

```
15 --Mark the level with the bidi tags
16 tagLevel :: Int -> [Level] -> [Ucs4]
17 tagLevel_[]=[]
18 tagLevel level ((x,y,z):xs)
19 |level/=x&& even x
20 =dirL++(map character ((x,y,z):xs)) ++dirEnd
21 |level/=x&& odd x
22 =dirR ++(map character ((x,y,z):xs))++dirEnd
23 |otherwise
24 =map character ((x,y,z):xs)
25
26 -- Mark the run with the bidi tags
27 tagRun :: Int -> Run -> [Ucs4]
28 tagRun z (LL xs)=parL++concat (map (tagLevel z)
29 (groupBy levelEql (mirror xs)))++parEnd
30 tagRun z (LR xs)=parL++concat (map (tagLevel z)
31 (groupBy levelEql (mirror xs)))++parEnd
32 tagRun z (RL xs)=parR++concat (map (tagLevel z)
33 (groupBy levelEql (mirror xs)))++parEnd
34 tagRun z (RR xs)=parR++concat (map (tagLevel z)
35 (groupBy levelEql (mirror xs)))++parEnd
36
37 --Insert mirror tags
38 mirror :: [Level] -> [Level]
39 mirror []=[]
40 mirror ((x,y,R):xs)
41 |isMirrored y
42 =(x,0xe004d,R):(x,0xe0049,R):(x,0xe0052,R):(x,y,R)
43 : minor xs
44 |otherwise=(x,y,R):(mirror xs)
45 mirror (x:xs)=x:(mirror xs)
```

[0123]

TABLE 18

Example Java Source Code for UniMeta

```
1 import java.util.*;
2 import java.io.*;
3
4 public class UniMeta {
5 BufferedReader dataIn;
6 String dirL="\udb40\udc44\udb40\udc49\udb40"+
7 "\udc52\udb40\udc02\udb40\udc4c"+
8 "\udb40\udc01",
9 dirR="\udb40\udc44\udb40\udc49\udb40"+
10 "\udc52\udb40\udc02\udb40\udc52"+
11 "\udb40\udc01",
12 dirEnd="\udb40\udc7f\udb40\udc44\udb40"+
13 "\udc49\udb40\udc52\udb40\udc01",
14 parL="\udb40\udc50\udb40\udc41\udb40"+
15 "\udc52\udb40\udc02\udb40\udc4c"+
16 "\udb40\udc01",
17 parR="\udb40\udc50\udb40\udc41\udb40"+
18 "\udc52\udb40\udc02\udb40\udc52"+
19 "\udb40\udc01",
20 parEnd="\udb40\udc7f\udb40\udc50\udb40"+
21 "\udc41\udb40\udc52\udb40\udc01",
22 mirror="\udb40\udc4d\udb40\udc49\udb40"+
```

TABLE 18-continued

Example Java Source Code for UniMeta

```
23 "\udc52";
24
25 String lBDO="<bdo dir=\"ltr\">",
26 rBDO="<bdo dir=\"rtl\">",
27 lP="<p dir=\"ltr\">",
28 rP="<p dir=\"rtl\">",
29 endP="</p>",
30 endBDO="</bdo>";
31 //Open the input file
32 public UniMeta(String in) {
33 try {
34 FileInputStream fileIn=new FileInputStream(in);
35 InputStreamReader str=
36 new InputStreamReader(fileIn, "UTF8");
37 dataIn=new BufferedReader(str);
38 }
39 catch (Exception e) {
40 System.out.printIn("Error opening file"+in);
41 return;
42 }
43 }
44 //Replace the unicode meta tags with HTML tags
45 private String replace(String in) {
46 StringBuffer out=new StringBuffer();
47 int i=0;
48
49 while(i<in.length()) {
50 if (in.startsWith(parL,i)) {
51 out.append(lP+lBDO);
52 i+=parL.length();
53 }
54 else if (in.startsWith(parR,i)) {
55 out.append(rP+rBDO);
56 i+=parR.length();
57 }
58 else if (in.startsWith(dirL,i)) {
59 out.append(lBDO);
60 i+=dirL.length();
61 }
62 else if (in.startsWith(dirR,i)) {
63 out.append(rBDO);
64 i+=dirR.length();
65 }
66 else if (in.startsWith(dirEnd,i)) {
67 out.append(endBDO);
68 i+=dirEnd.length();
69 }
70 else if (in.startsWith(parEnd,i)) {
71 out.append(endBDO+endP);
72 i+=parEnd.length();
73 }
74 else if (in.startsWith(mirror,i)) {
75 i+=mirror.length();
76 }
77 else {
78 out.append(in.charAt(i));
79 ++i;
80 }
81 }
82 return (out.toString());
83 }
84
85 //Process the input stream, generate output to stdio
86 public void parse() {
87 String in=null;
88 System.out.println("<html>");
89 try {
90 while ((in=dataIn.readLine()) !=null) {
91 System.out.println(replace(in));
92 }
93 }
94 catch(Exception e) {
95 System.out.println("Error parsing file");
96 return;
```

TABLE 18-continued

Example Java Source Code for UniMeta

```
97 }
98 System.out.println("</html>");
99 }
100
101 public static void main(String[]args) {
102 UniMeta input=new UniMeta(args[0]);
103 input.parse();
104 }
105 }
```

What is claimed is:

1. A method for providing metadata within a Unicode character streams said metadata describing information necessary for accurate display rendering of the character stream, said method comprising the steps of:

inserting one or more metatags into a Unicode character stream by spelling the tag identifier; and

inserting a metatag separator between multiple metatags if more than one metatag has been inserted, so as to create a modified character stream having separator-delimited metadata embedded within it.

2. The method as set forth in claim 1 further comprising the steps of:

inserting one or more parameters following at least one metatag with which it is associated; and

inserting a parameter separator between multiple parameters associated with a metatag if more than one parameter has been inserted so as to create a separator-delimited parameter list following a metatag.

3. The method as set forth in claim 1 wherein said step of inserting one or more metatags into a Unicode character stream comprises inserting an element metatag describes zero width joiner and zero width non joiner characters, such that multiple characters may be grouped together for treatment as a single grapheme or text element.

4. The method as set forth in claim 2 wherein said step of inserting one or more metatags comprises inserting a paragraph metatag, and wherein said step of inserting one or more parameters comprises inserting a right-to-left or a left-to-right directional parameter following a paragraph metatag which indicate a direction in which the character stream following the paragraph metatag and parameter is to be rendered for display.

5. The method as set forth in claim 2 wherein said step of inserting one or more metatags comprises inserting a direction metatag, and wherein said step of inserting one or more parameters comprises inserting a right-to-left or a left-to-right directional parameter following a direction metatag which indicate a direction in which the character stream following the direction metatag and parameter is to be rendered for display.

6. The method as set forth in claim 5 wherein said steps of inserting one or more metatags and inserting one or more parameters following metatags comprise the steps of replacing hyper text markup language bidirectional output (BDO) tags with said direction metatags and directional parameters.

7. The method as set forth in claim 2 wherein said step of inserting one or more metatags comprises inserting a mirror metatag which indicates the characters following the mirror metatag is to be presented in mirror fashion.

8. The method as set forth in claim 2 wherein said step of inserting one or more metatags comprises inserting a math metatag and a language metatag such that portions of the character stream which represent mathematical expressions are delimited from portions of the character stream which represent language.

9. A computer readable medium encoded with software causing a computer to perform the following actions for embedding display rendering metadata into character streams:

insert one or more metatags into a Unicode character stream by spelling the tag identifier; and

insert a metatag separator between multiple metatags if more than one metatag has been inserted, so as to create a modified character stream having separator-delimited metadata embedded within it.

10. The computer readable medium as set forth in claim 9 further comprising software for performing the following actions:

insert one or more parameters following at least one metatag with which it is associated; and

insert a parameter separator between multiple parameters associated with a metatag if more than one parameter has been inserted so as to create a separator-delimited parameter list following a metatag.

11. The computer readable medium as set forth in claim 9 wherein said software for inserting one or more metatags into a Unicode character stream comprises software for inserting an element metatag describes zero width joiner and zero width non joiner characters, such that multiple characters may be grouped together for treatment as a single grapheme or text element.

12. The computer readable medium as set forth in claim 10 wherein said software for inserting one or more metatags comprises software for inserting a paragraph metatag, and wherein said software for inserting one or more parameters comprises software for inserting a right-to-left or a left-to-right directional parameter following a paragraph metatag which indicate a direction in which the character stream following the paragraph metatag and parameter is to be rendered for display.

13. The computer readable medium as set forth in claim 10 wherein said software for inserting one or more metatags comprises software for inserting a direction metatag, and wherein said software for inserting one or more parameters comprises software for inserting a right-to-left or a left-to-right directional parameter following a direction metatag which indicate a direction in which the character stream following the direction metatag and parameter is to be rendered for display.

14. The computer readable medium as set forth in claim 13 wherein said software for inserting one or more metatags and inserting one or more parameters following metatags comprise software for replacing hyper text markup language bidirectional output (BDO) tags with said direction metatags and directional parameters.

15. The computer readable medium as set forth in claim 9 wherein said software for inserting one or more metatags comprises software for inserting a mirror metatag which indicates the characters following the mirror metatag is to be presented in mirror fashion.

16. The computer readable medium as set forth in claim 9 wherein said software for inserting one or more metatags comprises software for inserting a math metatag and a language metatag such that portions of the character stream which represent mathematical expressions are delimited from portions of the character stream which represent language.

17. A system for embedding metadata within a Unicode character stream, said metadata describing information necessary for accurate display rendering of the character stream, said system comprising:

a metatag inserter for inserting one or more metatags into a Unicode character stream by spelling the tag identifier; and

a tag separator inserter for inserting a metatag separator between multiple metatags if more than one metatag has been inserted, which creates a modified character stream having separator-delimited metadata embedded within it.

18. The system as set forth in claim 17 further comprising:

a parameter inserter for inserting one or more parameters following at least one metatag with which it is associated; and

a parameter separator inserter for inserting a parameter separator between multiple parameters associated with a metatag if more than one parameter has been inserted, which creates a separator-delimited parameter list following a metatag.

19. The system as set forth in claim 17 wherein said metatag inserter is adapted to insert an element metatag which describes zero width joiner and zero width non joiner characters, such that multiple characters may be grouped together for treatment as a single grapheme or text element.

20. The system as set forth in claim 18 wherein said metatag inserter is adapted to insert a paragraph metatag, and wherein said parameter inserter is adapted to insert a right-to-left or a left-to-right directional parameter following a paragraph metatag which indicate a direction in which the character stream following the paragraph metatag and parameter is to be rendered for display.

21. The system as set forth in claim 18 wherein said metatag inserter is adapted to insert a direction metatag, and wherein said parameter inserter is adapted to insert a right-to-left or a left-to-right directional parameter following a direction metatag which indicate a direction in which the character stream following the direction metatag and parameter is to be rendered for display.

22. The system as set forth in claim 21 wherein said metatag inserter and said parameter inserter are adapted to replace hyper text markup language bi-directional output (BDO) tags with said direction metatags and directional parameters.

23. The system as set forth in claim 18 wherein said metatag inserter is adapted to insert a mirror metatag which indicates the characters following the mirror metatag is to be presented in mirror fashion.

24. The system as set forth in claim 18 wherein said metatag inserter is adapted to insert a math metatag and a language metatag such that portions of the character stream which represent mathematical expressions are delimited from portions of the character stream which represent language.

* * * * *